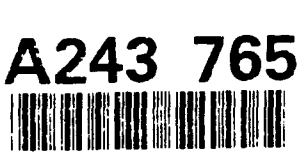


AFIT/GCS/ENG/91D-07

AN EVENT DRIVEN STATE BASED INTERFACE FOR SYNTHETIC ENVIRONMENTS

THESIS

Mark James Gerken
Captain, USAF

AFIT/GCS/ENG/91D-07

December 1991 Master's Thesis

A STATE BASED EVENT DRIVEN INTERFACE FOR SYNTHETIC ENVIRONMENTS

Mark J. Gerken, Captain, USAF

Air Force Institute of Technology
WPAFB OH 45433-6583

AFIT/GCS/ENG/91D-07

RL/COAA
Griffis AFB, NY 13441

Approved for public release; distribution unlimited

Previous research at the Air Force Institute of Technology (AFIT) has produced synthetic environments using head mounted displays (HMDs) and hand measurement devices. This thesis is a continuation of AFIT synthetic environment research.

Hand measurement data from a VPL DataGlove is combined with hand orientation and translational data to define a set of valid gestures. These gestures form the basis of an input language for class of finite automata known as finite state machines. Using the input symbols from this language, the user may define a finite state machine which associates a stream of input symbols with one or more system actions. The interface system developed is a combination of the *transition network* and *event* dialogue models described by Green and others.

The system is tested using a synthetic environment targetted to battlefield management. Defense Mapping Agengy (DMA) data is used to create a synthetic environment model of real world locations. Using the VPL DataGlove, the user of the synthetic environment can explore the landscape and populate it with objects common to a battlefiled such as aircraft hangers and anti-aircraft guns; the system provides the capability to generate and explore battlefields from within the synthetic environment.

Computer Graphics, Helmet Mounted Displays, Interactive Graphics

116

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL

AFIT/GCS/ENG/91D-07

AN EVENT DRIVEN STATE BASED INTERFACE FOR SYNTHETIC ENVIRONMENTS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

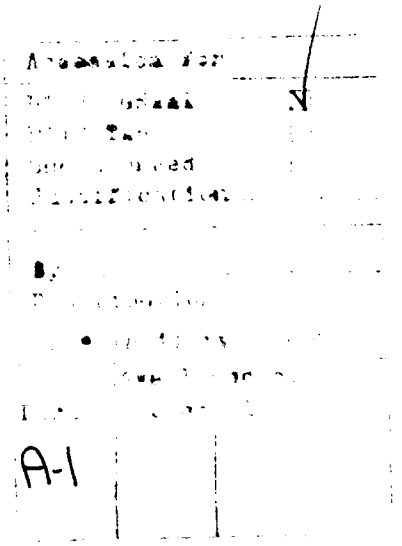
Requirements for the Degree of

Master of Science in Computer Engineering

Mark James Gerken, B. S.

Captain, USAF

December, 1991



Approved for public release; distribution unlimited.

Preface

The purpose of this study was to reasearch current synthetic environment interface systems, and to design and develop a synthetic environment interface system supporting the equipment available at the Air Institute of Technology.

I am indebted to several people for their assistance with this project. I would like to thank a fellow student, Capt John Brunderman, for his insights on manipulating synthetic objects. I would like to thank Capt William Watson for helping track down a communications problem with some of the hardware in the graphics lab. I also want to thank my thesis advisor, LtCol Phil Amburn, for his guidance; he was always available to answer my questions, and his comments and questions helped guide me during the development of this thesis.

I especially wish to thank my wife, Debora, for her patience and understanding.

Mark James Gerken

Table of Contents

List of Figures

List of Tables

Abstract

Previous research at the Air Force Institute of Technology (AFIT) has produced synthetic environments using head mounted displays (HMDs) and hand measurement devices. This thesis is a continuation of AFIT synthetic environment research. Specifically, a state based synthetic environment interface system is developed.

Hand measurement data from a VPL DataGlove is combined with hand orientation and translational data to define a set of valid gestures. These gestures form the basis of an input language for class of finite automata known as finite state machines. Using the input symbols from this language, the user may define a finite state machine which associates a stream of input symbols with one or more system actions such as deleting a selected synthetic object. The interface system developed is a combination of the *transition network* and *event* dialogue models described by Green (19) and others.

The system is tested using a synthetic environment targetted to battlefield management. Defense Mapping Agengy (DMA) data is used to create a synthetic environment model of real world locations. Using the VPL DataGlove, the user of the synthetic environment can explore the landscape and populate it with objects common to a battlefiled such as aircraft hangers and anti-aircraft guns; the system provides the capability to generate and explore battlefields from within the synthetic environment.

AN EVENT DRIVEN STATE BASED INTERFACE FOR SYNTHETIC ENVIRONMENTS

I. Introduction

1.1 Background

In 1965, Ivan Sutherland proposed the *Ultimate Display*:

> The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked. (35:508)

The *HoloDeck* aboard the mythical starship the *U. S. S. Enterprise* of *Star Trek: The Next Generation* fame is perhaps the most widely known example of Sutherland's *Ultimate Display*.[1] These multi-sensory, interactive computer systems, along with their computer-controlled objects and computer-defined behaviors, are known as *Virtual Worlds* (9:43) or *Synthetic Environments*.[2]

Although not nearly as sophisticated as those of the *HoloDeck*, synthetic environments are being built to address a diverse set of complex problems ranging from the interaction of drug molecules with target proteins (7), to battlefield command and control (1, 39, 20) and the visualization of military missions (44, 30).

---

[1] "The HoloDeck is a room that can create seemingly solid scenes and images, such as a jungle, a New Orleans bar, or Sherlock Holmes' sitting room, complete with simulated and seemingly intelligent characters." (46:78)

[2] Not all synthetic environments include equipment for multisensory interaction; many synthetic environments target only our sense of sight.

1.2 *Problem*

At the Air Force Institute of Technology (AFIT), synthetic environment researchers have focused their efforts on the definition and model of a synthetic environment (25, 23), techniques and tools for interfacing with a synthetic environment (29, 15), and development of a proof of concept of a synthetic environment targeted to mission planning (44). Although functional, these AFIT systems failed to take full advantage of the interface devices available:

- Only static finger postures (as measured by a VPL DataGlove [3]) were used to convey commands from the user to the system; moving finger gestures were not recognized. Further, the interpretation of the finger posture data was "hard coded" into the software; if a user wanted to use a different sequence of finger postures to trigger a system action than the sequence that was defined by the software, the software would have to be modified and recompiled.
- Data from the Polhemus 3-Space Tracker [4] was used only for positioning and orienting the DataGlove icon and the eyepoint in the synthetic environment. Both the orientation and trajectory of the DataGlove were ignored as possible sources of user command input.

In addition, the Polhemus and DataGlove device servers were implemented as read only servers; there were no provisions for dynamic reconfiguration of the user input devices. These shortcomings resulted in systems with a limited set of static commands that could be issued by the user from within the synthetic environment.[5] We needed a method to combine finger flexion values with the orientation and trajectory of the hand to create a richer language for communicating user desires to the synthetic environment. Futher, we desired a method which would support dynamic definition of the command sequences

[3]The VPL DataGlove uses fiber optics to measure the "flexion and extension of the fingers." (41)

[4]The Polhemus 3-Space Tracker uses magnetic fields to track the location and orientation of one or more sensors.

[5]It could be argued that an almost unlimited set of user commands could be recognized from just the flex angles of the five fingers. However, as noted by Sturman, Zeltzer, and Pieper, the inexactness of humans in replicating and maintaining finger positions results in the need for a fudge factor of 30-45 percent of the total finger flex value range in order to create a robust finger posture recognition system (34).

required to affect a system action (*i. e.*, a method that would not require a software modification to change the interpretation of the gesture sequences).

As noted by Sturman, Zeltzer, and Pieper, "hand measurement systems must sense both the flexion (flex) angles of the fingers and the orientation of the hand in real time in order to be useful input devices for virtual environments." (34:19)

1.3 Scope

The goal of my thesis was to create a state based, event driven whole-hand user interface to a synthetic environment. This interface system combines hand orientation and trajectory data with finger flexion values to create a rich set of synthetic environment whole-hand commands. For example, a finger posture can be used to indicate the user's desire to pick up and move a synthetic object, while the orientation of the head can be used to indicate the direction to move. In addition to combining the hand orientation and trajectory data with the finger flexion values to define the user commands, I implemented a state-based version of the whole-hand interface. Specifically, I defined a set of valid postures/gestures that may be issued from an initial state.[6] Based on the posture or gesture formed by the user, a new set of one or more postures/gestures is enabled. From a limited set of initial postures/gestures, a rich set of context sensitive command strings may be used.

I did not address the issue of finger posture recognition; instead, a commercial package from SimGraphics Engineering Corporation was used to control the VPL DataGlove and to recognized finger postures. The software system I developed supports:

- A Polhemus 3-Space Tracker to track the position and the orientation of the user's head and right hand (the image displayed to the user is appropriate for the user's orientation and position).
- A VPL Model 2 DataGlove to measure finger flexion values.

[6] This set is called a *language*, while the postures and gestures comprising the language are called its *alphabet*.

- The scaling, rotation, translation, duplication, and deletion of synthetic environment objects.
- Dynamic definition of the gesture sequences required to control and manipulate synthetic objects.
- The combination of finger flexion data with hand orientation and trajectory data to define user gestures.

The synthetic environment used to test the interface system is centered around the theme of battlefield management. Defense Mapping Agency (DMA) digital terrain data is used to construct terrain for the synthetic environment, and an initial set of force object icons is be provided.

To generate the illusion of depth, the raster images are generated using perspective transformations (*i. e.*, objects in the foreground are larger and move more quickly than like objects in the background).

1.4 Complimentary Efforts

Three separate, complementary thesis efforts at AFIT are used in support of my thesis effort:

1. Capt Brunderman's thesis effort resulted in a system for manipulating and rendering three dimensional PHIGS-like object models (8). I use Capt Brunderman's PHIGS-like object model and rendering pipeline for the objects in the synthetic environment.
2. Capt Simpson developed an object oriented flight simulator (32). In support of my thesis, I use the RS-232 port drivers, window manager, and queued input manager developed by Capt Simpson.
3. Capt Duckett implemented a method for sampling DMA digital terrain data (13); the sampled data is in a format suitable for rendering. I use the terrain models developed by Capt Duckett to provide realistic terrain for my synthetic environment.

1.5 Approach and Methodology

An object oriented analysis (OOA) and design (OOD) of the whole-hand interface was accomplished since, as noted by Booch (5:71), the object model offers several advantages:

- The use of the object model helps exploit the expressive power of all object-based and object oriented languages.
- The use of the object model encourages the reuse of not only software but of entire designs.
- The use of the object model produces systems that are built upon stable intermediate forms, and thus are more resilient to change.
- The use of the object model reduces the risk of developing complex systems, primarily because integration is spread out across the life cycle rather than occurring as one big bang event.
- The object model appeals to the workings of human cognition.

The system was developed through a series of prototypes, with each prototype adding capability to its predecessor.

The first step in the development of the whole-hand interaction system was the design and development of the software for controlling the VPL DataGlove and the Polhemus 3-Space Tracker. Once this software was implemented, the next step was to use one of the stations from the Polhemus to control the synthetic environment eyepoint, with a second station used to position and orient the DataGlove in synthetic space. The third step was to combine the VPL DataGlove finger flexion data with the Polhemus 3-Space hand position and orientation data to define and implement a set of whole-hand commands suitable for a synthetic environment. The final step in the evolutionary processes was be the design and implementation of the state based model for interpreting the gesture sequences as user requests for system action.

1.6 Materials and Equipment

Significant computational power is required to render and display synthetic environment images. Between 15 and 20 frames (completed, full-screen graphics images) per second must be rendered in order to preserve the continuity of movement within the synthetic environment (6). Due to the geometric complexity of the anticipated battlefield model, a special purpose graphics workstation is a cost effective candidate for hosting the synthetic environment software. Of the graphics workstations at AFIT, the Silicon Graphics IRIS 4D/310GTX is the most capable. This machine provides a hardware pipeline for rendering graphics images and supports two external RS-232 devices in addition to the display, keyboard, and mouse. The interface to the IRIS 4D/310GTX's computer graphics routines are supplied in several languages, including C. The object oriented version of C, C++, is therefore a logical choice for implementing the whole-hand interface system and is the language of choice.

1.7 Thesis Organization

The following chapter describes various synthetic environment interface devices and techniques, including the whole-hand interface technique. Chapter 3 describes the design of the whole hand interface system, including the design of the synthetic environment itself. Results and conclusions are discussed in Chapter 4.

II. Synthetic Environment Technologies

This chapter discusses some recent developments in synthetic environment interface equipment and techniques. Specifically, the following topics are addressed:

- Synthetic environment display systems.
- Tracking systems.
- Interface equipment, such as hand measurement equipment.
- The whole-hand interface technique.

2.1 Synthetic Environment Display Systems

Several types of display systems are used to support synthetic environments, ranging from standard video monitors to specialized head mounted viewing systems. Three classes of viewing systems are addressed in the following paragraphs:

- Standard video display terminals and screens, including large screen systems and single monitor stereo imaging systems.
- Head mounted displays (HMDs).
- Boom mounted displays (BMDs).

2.1.1 Video Display Terminals Video display terminals and screens can display high resolution images; the typical resolution of these systems is easily high enough to permit the use of textual displays to augment the graphics images. (In contrast, the relatively low resolution of HMDs makes impractical the use of textual information as a communications medium.) The high resolution of these display systems — as compared to the resolution of other synthetic environment display systems — may be necessary depending upon the application. For example, exploring a three dimensional representation of a database system from within a synthetic environment may require the display of textual information necessitating the use of a display system (such as the standard video monitor) that can legibly display text messages. An example system using this category of display

system is the GROPE III project developed at the University of North Carolina at Chapel Hill (UNC), where a large screen system is used to display rendered images of molecules (7).

Standard video monitors can also be used to display stereoscopic images. There are two ways to achieve stereo imaging using a single display screen. The first uses a polarizing screen and polarized glasses, while the second uses a lens shuttering system.

Polarized light systems alternately display images appropriate for the left and right eye (37). A polarizing shutter is attached to the display terminal and a pair of polarized glasses are used to view the images. Only the images appropriate for the right eye pass through the right lens of the polarizing glasses; the other images to the right eye (those that are intended for the left eye) are filtered by the glasses. Left eye images are similarly filtered. When run at a high enough rate, the stereo effect is achieved. The system described by Weimer and Ganapathy (45) uses this type of display system to display stereo images of the objects in their synthetic environment.

The second method of viewing stereo images using a single video screen also alternately displays left and right eye images. However, instead of using polarization to filter the images, a special pair of shuttering lenses are worn, such as the Crystal Eyes system from StereoGraphics Corporation (33). The Crystal Eyes system uses a transmitting unit to control the shuttering of the lenses so that only those images generated for the right (left) eye are seen by the right (left) eye.

Since two images are generated for each displayed frame (one image for each eye), two rendering engines are used to generate the graphics images. Almost any raster display system can be used to display stereoscopic images using the light polarization method (37). As a result, many of these display systems (depending upon the resolution of the monitor) can also legibly display textual information.

2.1.2 Head Mounted Displays

2.1.2.1 Origins of HMDs Sutherland built the first HMD for a synthetic environment while at Harvard University (36:43). His HMD displayed stereoscopic wireframe

images to the user through small CRTs; depth cuing was enhanced through image intensity modulation (images in the foreground were brighter than images in the background). Sutherland took his HMD with him when he moved to the University of Utah. In 1974, Vickers, at the University of Utah, improved Sutherland's HMD and added a pointing device to the environment (40). In 1976, Clark used the enhanced HMD to design free-form surfaces (11). Today, HMDs are widely used display devices for synthetic environment systems.

2.1.2.2 HMD Mechanics HMDs typically use two small CRT or LCD displays for displaying synthetic environment images.[1] Since two screens are used, HMDs can support stereo imaging: two sets of images can be rendered and displayed simultaneously, one for the right eye and one for the left eye.

The small screens in the HMD are mounted either vertically in front of the eyes (a closed or opaque viewing system (9)) or mounted horizontally above the eyes. The HMDs with vertically mounted screens use optics to increase the focal length, permitting the screens to be mounted closer to the eyes. Mounting the screens closer to the eyes not only decreases the moment arm of the helmet, but also increases the field of view (FOV) and consequently enhances the illusion of reality. HMDs with horizontally mounted screens use a mirror system to reflect the screens' images onto the focal plane of the eyes. These half silvered mirrors permit the user to view real world and synthetic environment objects simultaneously (called a see-through or an open viewing system (9)). Although the ability to view both real world and synthetic environment objects can be advantageous for some applications, open viewing systems can suffer from depth cue inconsistency (9):

- Unless the synthetic environment system knows the locations and shapes of the real world objects in the area surrounding the user of the synthetic environment, the synthetic environment objects will not be properly occluded by real world objects. For example, if a synthetic environment with an open viewing system were to have a synthetic ball roll under a real world table, the user of the synthetic environment

[1]Some HMDs use a projection system to display their images; CAE Electronics Ltd. of Quebec has developed such a system

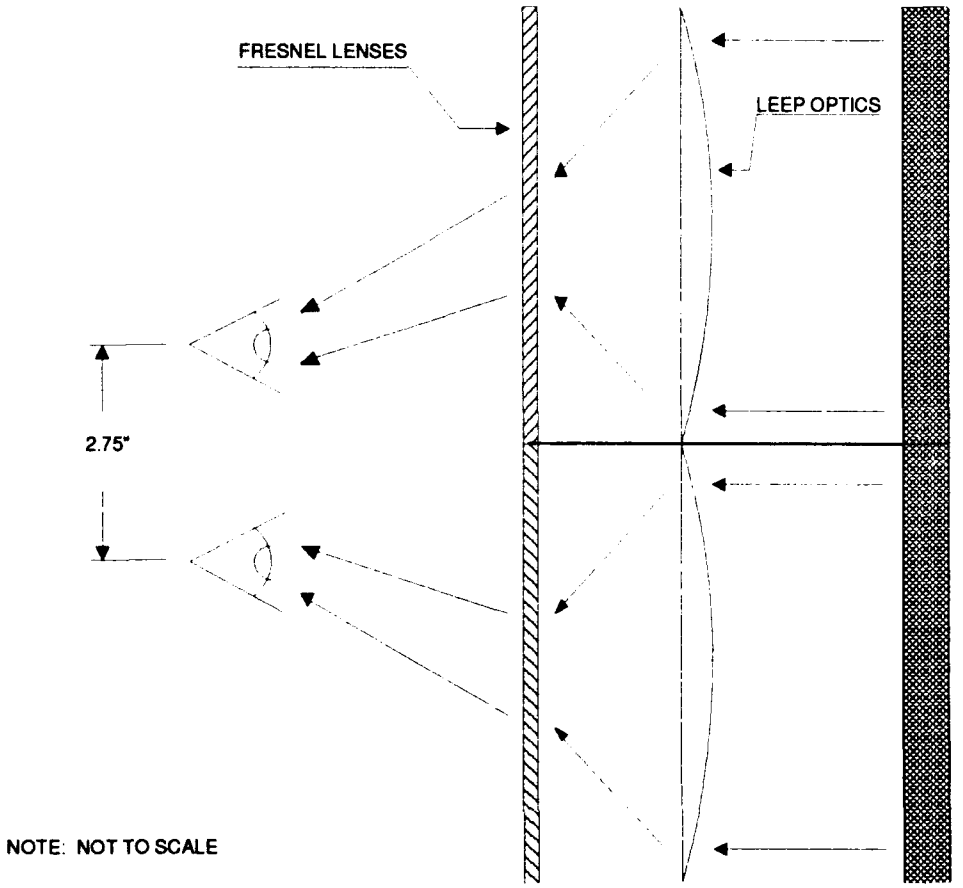


Figure 2.1. AFIT HMD Optics

will still be able to see the synthetic ball; the ball would not be occluded by the table (as it would be if it were a real ball), but instead it would appear to be on top of the table. Conversely, opaque synthetic environment objects will not completely occlude real world objects.

- There is no interaction between synthetic environment and real world lighting and shading models; the lights of one world do not cast shadows in the other world.

Although depth cuing may be problematic for open viewing HMD systems, open viewing systems provide the only means to interactively superimpose synthetic environment objects on real world objects. Consider a synthetic environment with a HMD open viewing system designed to aid aircraft mechanics, and suppose the mechanic was preparing to inspect the aircraft's wiring system. The synthetic environment system could project

schematics of the wiring system, and show the mechanic which panels to remove to inspect various wiring junctions. The open viewing system would allow the mechanic to overlay the synthetic environment schematics with the real world aircraft; this type of interaction is not possible from within a closed viewing system. In addition, open viewing systems place the user in familiar surroundings; this may help compensate for some system lag and thereby decrease the likelihood of a user becoming dizzy or disoriented (9). (System lag may cause some synthetic environment users to experience disorientation, especially in closed viewing systems.)

The synthetic environment system can track the position and orientation of the HMD to properly align the view volume in synthetic space (see Section 2.2 for more information on tracking systems). Wherever the user looks, the images he or she sees are appropriate for the user's position and orientation in the synthetic environment; the synthetic environment appears to totally surround the user.

2.1.2.3 Example HMDs UNC Chapel Hill has developed an open viewing HMD system for use with one of their synthetic environment systems (9). The HMD uses two liquid crystal color television screens for image display. The screens, measuring two inches on the diagonal, provide 220v by 320h pixels. Plastic lenses between the half silvered mirrors and the screens adjust the focal length. The HMD provides a 25 degree FOV horizontally, and the viewing apparatus is mounted in an adjustable helmet.

The AFIT HMD (29, 15) is a closed viewing system which uses two Sharp 3ML100 three-inch color LCD screens, and provides a 55 degree FOV at 240v by 360h pixels. The current AFIT HMD (15) incorporates LEEP optics to adjust the focal length to a comfortable distance, and uses Fresnel lenses to increase the apparent intraocular distance. (See Figure 2.1.) The use of the optics distorts the image somewhat (15:41), but the wide angle optics enhance the visual effect (9). A modified bicycle helmet is used to house the optics and viewing screens. The AFIT HMD is shown in Figure 2.2.

Higher resolution HMDs are available commercially. For example, the VPL EyePhone HRX provides just over 300,000 primary color pixels (42).

HMDs provide one of the most powerful kinetic feedback mechanisms available to

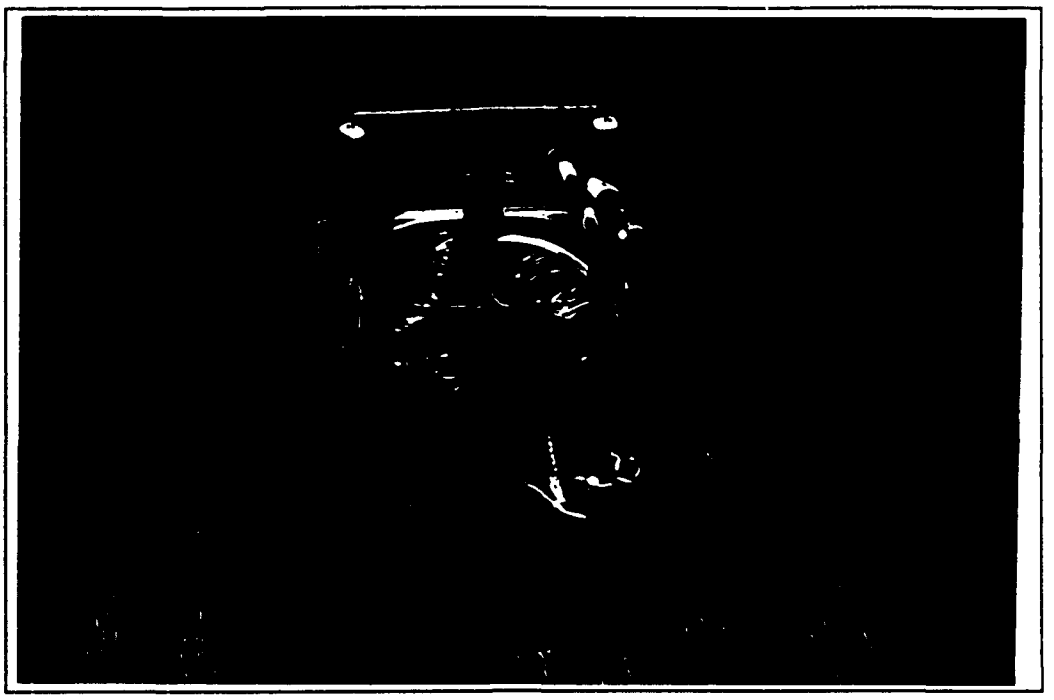


Figure 2.2. The AFIT Head Mounted Display

synthetic environment programmers: synthetic objects can be moved out of sight as the user turns his or her head (9). In addition, many HMD systems support stereo imaging: a separate image can be displayed on each screen of the dual screened HMDs.[2] HMDs allow the user to change his or her viewpoint by the most natural way possible: by walking toward and examining the synthetic object (21). For a more detailed introduction to HMDs, see (16) or (9).

2.1.3 Boom Mounted Displays BMDs are similar to HMDs, except that the method of mounting and tracking the video screens differs. Instead of mounting the video screens in a helmet worn by the user, BMDs mount the screens in a viewing apparatus located at the end of a boom or arm. Since the boom or arm is used to support the weight of the screens, larger, higher resolution video screens may be used; a boom mounted display system is depicted in (16). Due to the method of mounting the displays, the orientation

[2]The ability of an HMD to display stereo images is not determined solely by the presence of two video screens: "It is not true that binocular or stereoscopic vision requires two distinct pictures (one for each eye) — what is required is the presentation of two separate views of one object space (one view for each eye)." (10:79)

of the user's head may be determined from flex sensors mounted on the boom or arm. In this respect, the method of tracking the orientation of the user's head to determine the appropriate synthetic environment viewpoint differs from that used in HMD systems.

2.2 Tracking systems

If a synthetic environment system is to allow the viewpoint to be slaved to a HMD system, a means to track the position and orientation of the HMD must be used. In Sutherland's system, the orientation and position of the HMD was tracked using either ultrasonic or mechanical linkage systems (9). There are three primary methods for synthetic environment systems to track the position and orientation of real world objects:

- Mechanical linkage systems.
- Optical tracking systems.
- Magnetic tracking systems.

2.2.1 Mechanical Linkage In a mechanical linkage system, a boom or arm is used to measure the position and orientation of an object, be it a HMD or some type of input device. The GROPE III system at UNC Chapel Hill uses a mechanical arm to track the position and orientation of the hand (7). The mechanical arm not only has the advantage of jitter free measurement, but it also facilitates another form of synthetic environment interaction: force feedback (see Section 2.3.2). Mechanical linkage systems sacrifice mobility for increased accuracy, jitter reduction, and force feedback potential. Other means of tracking the user's head position and orientation may be less intrusive than a mechanical linkage system, but the jitter free position and orientation data results in a stable viewing environment.

2.2.2 Magnetic Tracking Systems Magnetic tracking systems use magnetic fields to track the position and orientation of one or more sensors. When a sensor moves through the field, electrical current is produced. The tracking unit can measure the current levels from the sensor to determine its position and orientation. Depending on the capability of the tracking unit, one or more sensors may be used.

There are several advantages of using a magnetic tracking system:

- Relatively low cost. Magnetic tracking systems are relatively inexpensive when compared to commercial optical tracking systems.
- Increased mobility. The user is free to move about as he or she pleases; he or she is limited only by the range of the source and the length of the data cable from the sensor to the tracking unit.
- Clothing will not occlude the sensor. The sensor does not need to be on a direct line of sight with the source.

However, magnetic tracking systems are prone to noise (34, 9, 2). Metal objects, electrical wiring, and electrical equipment (including computer monitors) can warp the magnetic field; the field distortion will manifest itself in the measured data (3). Also, as the distance between the source and the sensor increases, accuracy decreases. Chung, *et. al.* (9), reports that their graphics lab contains "many ...sources of magnetic perturbations", including a metal floor, which warp the magnetic field of the tracking unit such that "orientation values reported close to the Polhemus' spacial range are off by as much as 30 degrees." The data inaccuracies may cause problems for the developers of the synthetic environment: the displays of magnetically tracked HMD systems tend to exhibit jitter unless some type of filtering of the HMD position and orientation data is performed. Sturman, *et. al.* (34), reported that they attempted filtering the data from their magnetic tracking system, but found that the filtering process introduced an unacceptable system lag; they found a fast but noisy tracking system preferable to a slow but smooth tracking system.

An example magnetic tracking system is the *Flock of Birds* system from Ascension Technology (2). This system can track up to six receivers as they move through the magnetic field. The tracking radius of the system is six feet; but an optional upgrade can expand this distance to 16 feet. Each receiver is sampled at a 100 Hz rate; the data can be reported back to a host computer via an RS-232C link running from 2400 baud to 115,200 baud. Multiple source units can be employed, further extending the tracking range. The reported angular accuracy of the *Bird* tracking system is 0.1 degree at eight feet. Another

popular magnetic tracking system is the Polhemus 3-Space Tracker (for more information about the Polhemus 3-Space tracker, see Section 3.6).

2.2.3 Optical Tracking Systems Optical tracking systems combine the freedom of movement of the magnetic tracking systems with the accuracy of the mechanical linkage systems; optical tracking systems offer magnetic noise immunity. These tracking systems use light sources and photosensitive receivers to measure orientation and position data. Optical tracking systems can be configured in one of two ways:

1. The light sources (small LEDs in the case of the GEC system (18)) can be mounted on the device to be tracked, while the static lens system is used to determine the position and orientation of the tracked device.
2. The light sources may be mounted on stationary objects in real space, while the photoreceptor is mounted on the device to be tracked. This configuration is known as "inside out" tracking (4).

The light sources either flash in a pattern that identifies their absolute location on the device or are arranged in a predetermined pattern that the tracking system can use to identify the orientation of the device. A sensor unit receives the flashes of light from several of the sources. Based on the known firing sequence or pattern of the light sources, coupled with the time delta from the firing of the light to the reception of the light by the photoreceptor, the tracking system is able to calculate the position and orientation of the device. The only disadvantage of optical tracking systems, besides their relatively high cost, is that the LEDs or the photoreceptor can be occluded by clothing or other real world objects; the tracker can lose sight of the tracked object.

An example optical tracking system is the GEC Ferranti Corporation's GRD-1010 (18). This optical tracking system mounts a series of LEDs on the device to be tracked and uses a predetermined firing pattern for LED identification. Using the time delta from the firing of a known LED to the reception of the LED's light at the photoreceptor, coupled with the known absolute position of the LED on the device, the optical tracking system is

able to calculate the position and orientation of the device; the orientation data is accurate to within one milli-radian. The system operates at a 240 Hz sample rate.

2.3 Interface Equipment

Sutherland recognized the need for complete sensory input to create the illusion of reality. As noted by Chung, *et. al.* (9:42):

> Most important is kinetic feedback — the response of the computer display the user's movement. The senses of sight, sound and feeling lend themselves most easily to this effect, as objects can be moved out of sight, apparent sound sources can change their relative position when the user's head is turned, and force feedback mechanisms can respond to hand and arm movements.

HMDs are effective visual interfaces to synthetic environment systems, but HMDs alone cannot provide more than a visual interface to the system; other interface equipment is required. Sound systems, haptic feedback systems, and hand measuring systems have been developed to provide enhanced kinetic feedback to the users of synthetic environments, and to provide a means for user command input from within a synthetic environment.

2.3.1 Sound Systems There are two classes of auditory feedback provided by synthetic environment systems (9):

1. Clicks and clacks. This level of auditory feedback is the simplest to implement, and is typically used to acknowledge a user command input (e. g. , the synthetic environment may respond with a click cue following the recognition of a hand gesture), and to provide contact cues (such as a "clack" when two pool balls collide on a pool table).
2. True 3D sound where the sound is located outside the head at some discrete distance and location. (See (47).)

Synthetic environment sound systems are not limited to sound generation; some synthetic environment systems use voice recognition subsystems to provide a verbal interface.

For example, a synthetic environment developed at AT&T Bell Labs uses an AT&T VR1 voice recognition subsystem to recognize up to 40 two second long phrases. From within the AT&T synthetic environment, a user can rotate, scale, and translate synthetic objects using "relative index finger position or hand rotation. These transformation tasks are invoked by saying: *translate*, *rotate*, or *scale*." (45:238)

2.3.2 Haptic Systems Haptic systems attempt to communicate with us through touch, temperature, pressure, *etc.* These systems attempt to have the users of the synthetic environment feel the textures and forces of synthetic objects. As noted by Minsky, *et. al.*, (26:235):

> Force display is especially useful for communicating surface texture and bulk properties of objects and environments as well as dynamics of objects ...[F]orce display technologies augment the strengths of computer generated graphics and sound in creating convincingly realistic environments.

Haptic feedback is generally accomplished through one of three methods:

1. Mechanical linkage systems, were mechanical actuators attached to specialized I/O devices convey force and texture data to the user, such as in the *Sandpaper* system (26).
2. Pneumatic systems. Air bladders in specialized clothing inflate giving the user some sense of contact (via pressure) with a synthetic object. Pneumatic systems can convey only force levels; no texture information can be communicated to the user. The VPL DataGlove THX is an example of this type of feedback device (43).
3. Simple vibrators. Vibrators can only provide an indication that some type of contact has been made with a synthetic object; no force information can be relayed to the user through a simple vibrator. A simple vibrator provides enough feedback (in addition to the visual cues) to enable the user to grasp synthetic objects.

An example mechanical linkage system is the Argone Remote Manipulator (ARM) used in the GROPE III project at UNC (7). The GROPE III ARM is used in the exploration of molecular docking problems: molecular bond forces of synthetic environment

molecules are conveyed to the user through the ARM. The forces exerted by actuators in the ARM are proportional to the forces that would be experienced by the molecule. For example, if a molecule controlled by the ARM were to be repelled by the target molecule, the user would find it increasingly difficult to physically move the ARM (and hence the molecule it controls) toward the target molecule. Molecular biologists use this system to investigate the relative effectiveness of molecular compounds (7). Brooks, *et. al.*, found that the addition of force feedback to their system "radically improved situational awareness." (7:177)

Haptic systems need not be complicated. For example, another system at UNC, *A Mountain Bike with Force Feedback for Indoor Exercise* (38), allows a user to climb aboard a physically stationary bike and ride across a synthetic countryside. The rear wheel of the bike is attached to a resistance device that controls the difficulty of peddling. When riding across flat synthetic terrain, little effort is required on the part of the user to keep peddling. However, as the user begins to climb hills, the system increases the pressure to the bike braking system, requiring the user to peddle harder to maintain his speed; the haptic system allows the user to "feel" the force of gravity affecting the bike.

At least one author has reported on the necessity of force feedback mechanisms for the direct manipulation of objects in synthetic space. Weimer, *et. al.*, (45:237) notes that

> ...without tactile feedback, it is impractical to try wrapping the fingers around objects as we would in the real world ...Tactile feedback would be critical if the user wants to reach into the remote site to establish grasps for a manipulator.

2.3.3 Hand Measurement Equipment Hand measurement devices measure the finger flexion values using either fiber optics (as in the case of the VPL DataGlove series) or using hall effect sensors (as in the case with the Exos Dexterous Hand Master). Some sort of tracking equipment is used to position and orient the user's hand in synthetic space. (Note that some hand measurement devices are haptic devices as well; for example, the VPL DataGlove THX reads finger flexion values and provides a pressure feedback mechanism.)

The finger flex values can be used to issue commands from within the synthetic environment. For example, suppose that the hand measurement device we are using returns five flex values (one per finger). The flex values could be used as a vector into a look-up table containing the minimum and maximum flex value for each sensor for a given posture. The posture data can then be interpreted as a request for system action. In

Finger Flexion Data										Corresponding
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Posture
0	20	0	20	0	20	0	20	0	20	Palm
60	90	60	90	60	90	60	90	60	90	Fist
0	20	60	90	60	90	60	90	60	90	Thumb

Table 2.1. Sample DataGlove Posture Table

Table 2.1, three postures are supported. To issue a command from within the synthetic environment, the user need only bend his or her fingers to match the data values for the posture corresponding to the desired command. The values in the look-up table should be determined by experimentation (34); both a minimum and maximum value are included to compensate for the inexactness of humans in forming and maintaining finger postures (see (34)).

Two commercial hand measurement devices are the VPL DataGlove and the Dexterous Hand Master by Exos:[3]

- The VPL DataGlove Model 2 (41) is a nylon glove with fiber optic cables attached to the back of the glove; the cables measure the bending angles of the thumb and of the lower and middle knuckles of the other fingers. As the fingers bend, they in turn bend the cables. The light escaping from each bent cable is directly proportional to the flexure angle. Photoreceptors measure the attenuated light signals and calculate the flexure values. The glove can be sampled at a 60 Hz rate, with the data transmitted via a serial communications line to the host computer. The VPL DataGlove is shown in Figure 2.3.
- The Dexterous Hand Master form Exos is an exoskeleton device that uses hall sensors to measure 15 flexion values (three per finger) and five radialulnar deviation values

[3]Other hand measurement devices exist. For example, see (22).

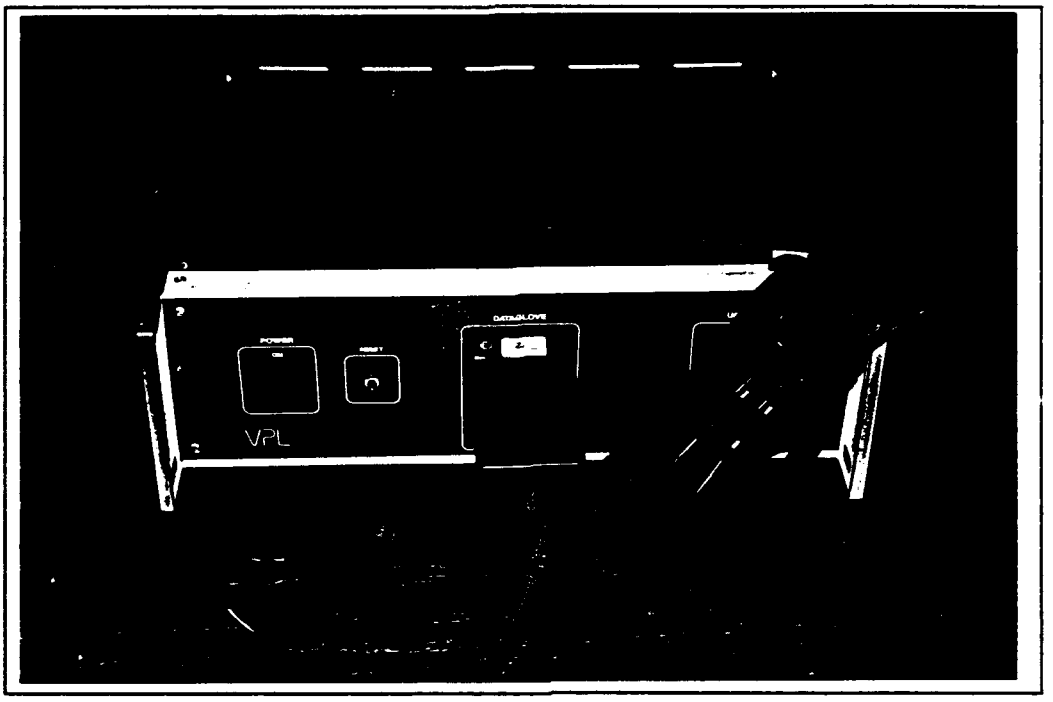


Figure 2.3. The VPL DataGlove

(side to side motion). This device can measure the flexure angles to within $\frac{1}{2}$ degree (the highest accuracy of any hand measurement device currently on the market (14)). The Dexterous Hand Master is shown in Figure 2.4.

Note that the flexion values need not be used simply as a vector into a look up table; the flexion values themselves may be used as a valuator input. For example, the *edvol* system described by Kaufman, *et. al.* (24), uses the bending of a finger to control the relative scaling amount for a selected object. Also, the orientation of the hand may be significant in identifying the meaning of a finger posture[4] (consider the difference between a "thumbs up" and a "thumbs down" gesture). Sturman, *et. al.* (34) has expanded this concept into the whole hand interaction paradigm.

[4] I have adopted the terminology used by Sturman, *et. al.* (34). Postures are stationary hand or finger positions, while gestures involve hand or finger motion.

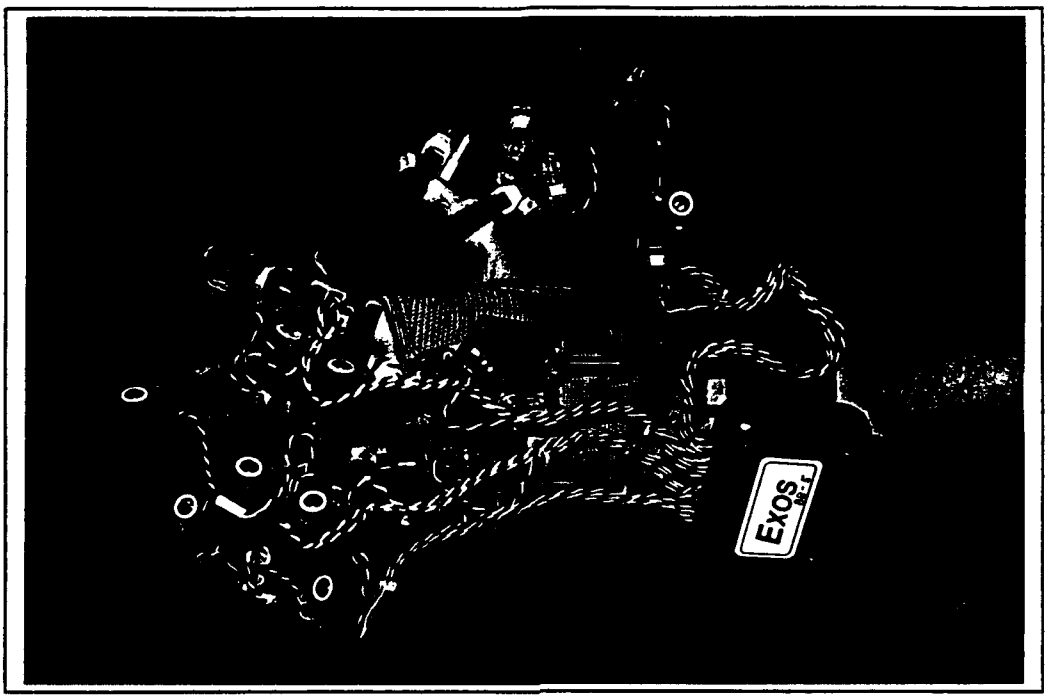


Figure 2.4. The Dexterous Hand Master

2.4 The Whole Hand Interface Technique

The *bolio* system at the MIT Media Lab was developed to "assess the value of whole-hand interaction for direct 3-D manipulation of virtual objects and as a command input device. " (34:19) From the *bolio* project, the whole-hand interface paradigm has evolved.

The MIT Media Lab has identified three ways of approaching whole-hand interaction (34):

1. Direct manipulation, where the user manipulates synthetic objects as he or she would manipulate real objects (*i. e.*, pick them up and move them, *etc.*).
2. An abstracted graphical input device such as a button, a valuator, or locator.
3. As a stream of tokens in some language.

The idea of the whole-hand interface is to combine the position and orientation of the hand with the finger flexion data to create a rich set of user commands. For example, the system described in (24), the *edvol* system, uses a thumbs up to signal approval, and

a thumbs down to signal disapproval; both commands are oriented postures, where the orientation of the hand determines the intended meaning. The concept is not limited to static hand orientation with static finger postures; moving hand and moving finger commands are possible. The MIT Media Lab has identified nine ways of combining hand and finger data to form user commands as shown in Table 2.2 (34:21).

Hand Position & Orientation	Finger Flex Angles		
	Don't Care	Motionless Fingers	Moving Fingers
Don't Care	X	Finger Posture (button) e. g. fist	Finger Gesture (valuator)
Motionless Hand	Hand Posture	Oriented Posture (e. g. thumbs up)	Oriented Gesture (e. g. come here v. good bye)
Moving Hand	Hand Gesture	Moving Posture (e. g. banging fist)	Moving Gesture (e. g. strong come here)

Table 2.2. Hand Motion Taxonomy

The lower right portion of Table 2.2 represents the most difficult gestures to recognize. Moving gestures (such as the strong come here) are subject to sampling problems. If the hand measurement device is sampled once per frame, critical portions of the gesture could be missed, especially if the user moves into a geometrically complex portion of the synthetic environment. (The higher the geometric complexity of the scene, the lower the frame rate, and hence the lower the sampling rate.)

The use of motion is context sensitive (34:22); not every twist of the wrist or every translation of the hand is interpreted as a user command. As discussed in the next chapter, I use the results of the *bolio* project to design and develop a state-based whole-hand synthetic environment interface system.

2.5 Summary

This chapter has described some of the techniques used to interact with synthetic environments, including the whole-hand interface technique and the necessity of tactile feedback for direct manipulation of synthetic objects. The following chapter describes the development and design of the state based interface system.

III. System Design

This chapter describes the design of the whole-hand synthetic environment interface system as implemented in the BattleManager program. Included is a discussion of the design constraints imposed on the design of the BattleManager system, an assessment of the reusability of the existing AFIT synthetic environment software as it applies to the object oriented BattleManager system, and an elaboration of some of the design decisions made during the development of this system. Prior to system design, a domain analysis was performed to discover the objects in problem space. The results of this domain analysis are presented in Appendix A.

3.1 Design Constraints

Several design constraints were taken into consideration during the system design phase:

- AFIT's version of the software from the SimGraphics Engineering Corporation for controlling the VPL DataGlove does not correctly execute on a Silicon Graphics 4D machine (nor was the software designed to do so); the DataGlove software must be remotely hosted.
- The AFIT graphics lab does not have the equipment necessary to provide tactile feedback.
- The AFIT HMD system must be supported.
- The interpretation of the gesture stream formed by the user must be under user control; a means to allow the user to identify a stream of one or more gestures with a corresponding system action must be supported.

The lack of both audible and tactile feedback mechanisms has a significant impact on the system design:

- No direct manipulation of synthetic objects can be supported; the system must provide a means to allow the user to indirectly select and modify synthetic objects.

This constraint is satisfied by defining system actions for highlighting and selecting synthetic objects for manipulation based solely on gesture and posture data; the user does not need to grasp the objects of interest. The the whole hand interface system therefore treats the gestures and postures formed by the user as a stream of tokens in a defined language and, depending on the system action invoked, the data from the DataGlove and the Polhemus may be treated as an abstract graphical input device such as a valuator. (See Section 3.7.)

- The only feedback to the user's gesture stream can be visual; in support of the AFIT HMD, this visual indication must not include text. This constraint is satisfied by changing the color of the synthetic object (including the DataGlove icon) in response to user requests for system action.

Section 3.3 discusses the design satisfying the above design constraints. However, before any new software was developed, the existing AFIT synthetic environment software was analyzed for its potential reuse in the BattleManager system. The results of this analysis are presented below.

3.2 Analysis of the Existing AFIT Synthetic Environment Software

Software for controlling the VPL DataGlove and the Polhemus 3-Space Tracker was developed under a previous AFIT thesis effort (15); the software for controlling the DataGlove incorporates the commercial package from SimGraphics Engineering Corporation. Each component of the software for controlling the DataGlove and the Polhemus devices was written in the C language. These programs were designed to run on a DEC MicroVax, while the main synthetic environment program was designed to execute on a graphics workstation. Communication between the existing synthetic environment package (the VE program) and the package controlling the Polhemus and the DataGlove took place through a UNIX socket, while communication between the interdependent device servers took place through a shared memory segment. During system initialization, the device servers would be spawned from a single program: the VEserver. The VEserver was responsible for determining which devices were active and was responsible for controlling

— via a clone of the VEserver — communications with the client. Several user interface devices were supported, but only the software for controlling the Polhemus and the DataGlove are of interest (the other devices are not needed to support the state based whole-hand interface system). The structure of the VEserver package is shown in Figure 3.1.

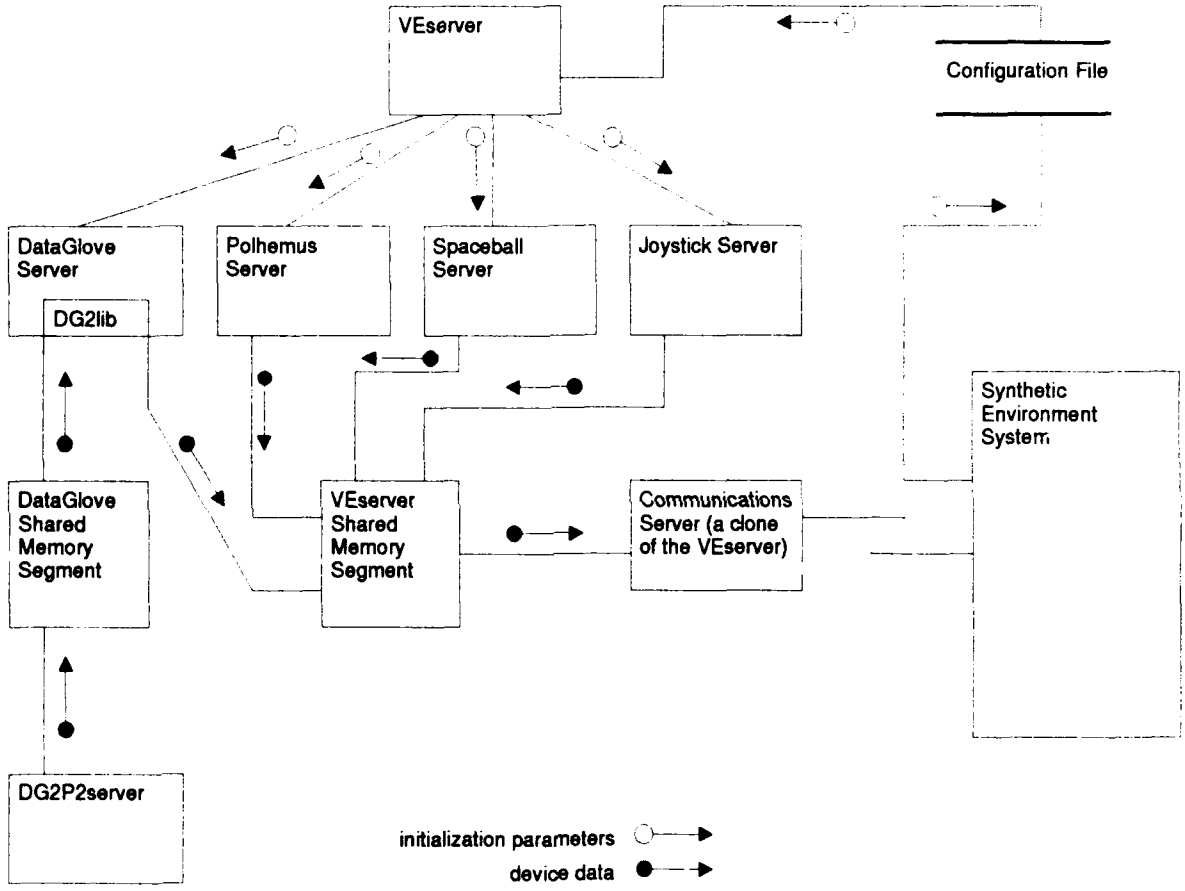


Figure 3.1. The Existing AFIT Synthetic Environment System.

3.2.1 The SimGraphics Library As shown in Figure 3.1, the SimGraphics library of DataGlove routines — DG2lib — interfaces with the DG2P2server program, also a SimGraphics product. The DG2P2server encapsulates the interface to the VPL DataGlove, and is responsible for communicating with the DataGlove device. The DG2P2server pro-

gram contains the code for recognizing the distinguished finger postures.[1] To aid the recognition of finger postures, the DG2P2server contains a table of distinguished finger postures similar to Table 2.1. A total of 25 distinct finger postures is supported by the DG2P2server program.

When the DG2P2server program is initiated, it obtains 4,668 bytes of shared memory. The shared memory identifier and the address of the shared memory block are made available to the DG2lib routines. Following the shared memory operations, the DG2P2server waits for an application program to attach to the shared memory block via the DG2lib routine *DG2_open*. Only one DG2P2server program can be active on the system at any one time; the DG2P2server program checks for the existence of another DG2P2server program before obtaining a shared memory block. If another DG2P2server program is active, the second DG2P2server program will abort with an error message.

Once initialized, the DG2lib routine *DG2_get* reads a data packet from the DataGlove device and returns the *DG2glove* data structure shown in Table 3.1. If the user's finger postures do not match those of a distinguished posture, the *posture_name* field is NULL and the *posture_number* is zero.

```
typedef   struct {
          float          x, y, z;              /* IsoTrak position in inches */
          float          yaw, pitch, roll;     /* Euler angles in degrees */
}         DG2_spacial;

typedef   struct {
          DG2_spacial    spacial;              /* IsoTrak position & orientation*/
          short          flex[10];             /* finger flexion data */
          char           posture_number;       /* the posture number */
          char           posture_name[16];     /* the posture name */
}         DG2glove;
```

Table 3.1. DG2lib DataGlove Record Format

In support of the AFIT HMD, the BattleManager program uses the VPL DataGlove as a source of user input and therefore requires the ability to control the DataGlove and

[1] By distinguished finger postures I mean the set of finger postures defined and named by the software developer. The set of distinguished postures is created through SimGraphic's DG2Gest program.

recognize distinguished finger postures. The SimGraphics packages provide this capability and are therefore used to support of the BattleManager system. The SimGraphics DG2P2server program was modified to support UNIX signals (see Section 3.5.2.3), but was otherwise completely reusable. However, the remaining VEserver components were not as reusable.

3.2.2 The Server Programs In the existing system, the server programs SpaceballServer, JoystickServer, DataGloveServer, and PolhemusServer [2] (hereafter collectively referred to as the server programs) are slaved to the VEserver program. All client requests for device data come through the clone of the VEserver program where they are immediately satisfied from a shared memory block. All server programs are limited to only operations: Each server program cycles through an endless loop reading their respective user interface devices and writing the resulting data packets into the VEserver's shared memory block where they can be accessed by the communications server. Once initialized, the VEserver program operates as follows (15):

```
begin
    parse the configuration file for active devices;
    obtain a shared memory block for the device data;
    for each active device do
        spawn the appropriate server;
    obtain and initialize a UNIX socket;
    bind to the socket and listen for connections;
    loop
        accept a connection to the socket;
            /* accepting a connection is a blocking operation */
        clone the VEserver;
        if we are the clone then service the communications link;
            /* the clone will terminate servicing the communications link */
```

[2] Filer (15) called the server programs the VE_SB_server, VE_JOY_server, VE_DG_server, and VE_POL_server respectively.

```
    end loop
end
```

The configuration file parsed by the VEserver (the master configuration file) contains the names of the active I/O devices; a separate configuration file for each device identifies the RS-232 port number and communication parameters to use for the device. If the DataGlove is an active device, the master configuration file may specify the names of the calibration file and posture file to be used instead of the VEserver's default calibration and posture files. The socket obtained by the VEserver program is initialized for a network port number identified in the C header file of the VEserver program; changing the port number requires a modification to the source code. If any error occurs during the spawning of the device servers, the initialization of the socket, or during the creation of the shared memory block, the VEserver will abort.

3.2.2.1 Suitability of the Existing Server Programs The implementation details of the device servers cannot be cleanly encapsulated by the client side software:

- Client side objects that implement an interface to a server side I/O device[3] are restricted in the methods (functions) they may provide due to the read only nature of the device servers. For example, the client side object for the Polhemus would not be able to provide functional methods for defining the alignment reference frame or for performing a sensor alignment (boresight); the client side objects would be tightly coupled to the VEserver's implementation as a read only server.
- Since all the device servers are spawned at the same time, no client side object encapsulating an I/O device may contain code for opening or initializing the device as part of the object's constructor. The constructors for the client side I/O devices may only activate a device by writing the appropriate information into the VEserver's configuration file before the VEserver program is initiated.

[3] I/O devices could be connected to and controlled by the client.

The current set of VEserver programs supports only one user per system run since the only way to specify the calibration file for the DataGlove is through the configuration file. Supporting a series of users requires that the VEserver allow dynamic loading of the DataGlove configuration file. With the existing set of server programs, the only way to support a series of users is to terminate and restart the VEserver programs or to force the new user to use the calibration file from the previous user. (If the new user's hand size and dexterity differ greatly from that of the initial user, the software may be unable to recognize the distinguished finger postures formed by the new user.) Either case is unacceptable.

There are two ways to overcome these deficiencies:

1. Modify the server programs to allow dynamic device control, including independent generation of the server programs by the VEsever and the capability to alter the operating characteristics of the devices after the server programs have been initiated.
2. Replace the server programs with a newly developed set of independent device servers.

Modify the Existing Server Programs As shown in Figure 3.2, the existing set of server programs could be modified to support dynamic creation and destruction of the various device servers, and could be further modified to support dynamic control over the devices themselves. However, implementing these modifications would require a significant restructuring of the server programs. In addition, one artifact of the structure of the server programs would remain: there is only one communications path from the device servers to the client, requiring a client side object to encapsulate and control communications with the server.

If the client does not provide an object for controlling the single link with the server, problems could arise:

- Data or status sent from a device server could be read by the wrong client object. The client objects should have no knowledge of each other's existence otherwise they would share control information[4]; without the knowledge of each other's existence,

[4]Constatine and Yourdan call this type of coupling *control coupling*.

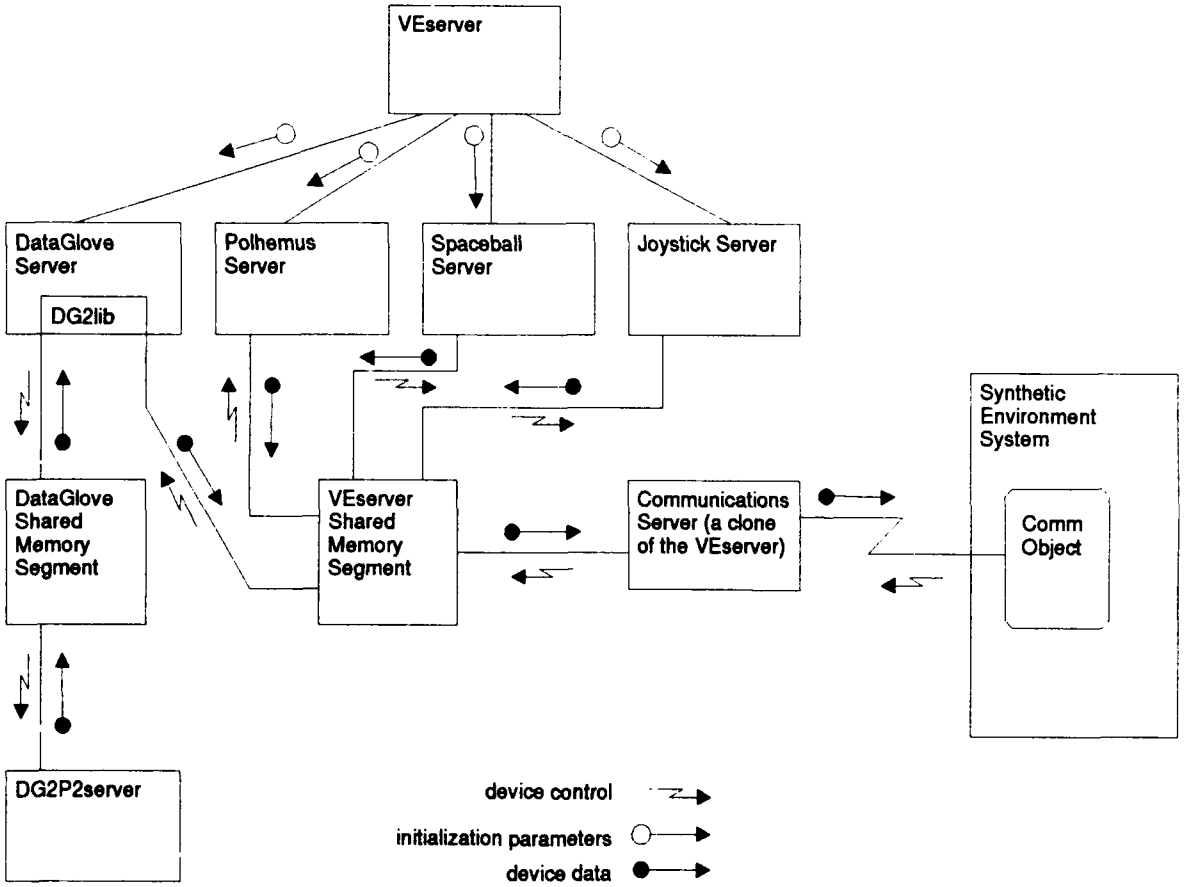


Figure 3.2. Hypothetical Device Server Program Structure.

the objects would only have knowledge of the commands they themselves had sent to the server. Each client side object would anticipate that the data coming across the socket connection would be in response to a command they had issued; this assumption would be violated if each client side object were granted direct access to the single communications link.

- The client side control logic could be constructed so that method calls to objects encapsulating a device were made in such an order that no two objects would simultaneously be expecting data from the server. This would result in the client's control logic being tightly coupled with the single communication line implementation of the server. In addition, the required ordering of client side method calls could very well

result in a non-optimum implementation.

Providing multiple communications paths from the server to the client (one or more communications path per device) could eliminate the above communications problems, but would require an even greater modification to the device servers and calls into question the wisdom of using the existing set of VEserver programs as a basis for achieving the design goals of loosely coupled, fully functional client side objects.

Develop New Device Servers Development of a new system of device servers would satisfy the BattleManager's requirement for loosely coupled dynamic device interface objects. A newly developed set of independent device servers offers many advantages over modifying the existing implementation:

- The server's shared memory block for the devices is no longer needed since each device can create its own buffer for building and storing device data records; the server programs would be decoupled from each other.
- The communications bottleneck between the client and the server would be greatly reduced since each device would transmit its data and status over a dedicated communications link.
- The client's device controller objects could be loosely coupled (i. e. independent of each other). In addition, there would be no need for a single client side communications object to control communications with the server; there would be no possibility of a client object reading the data meant for a different client object if each client object contained its own dedicated communications link. A general class for creating and controlling UNIX sockets could be developed and incorporated into each of the client's device interface objects.
- The client's control loop would no longer have to be concerned with ordering method calls to ensure that only a single device was prepared to transmit data; the client's control loop would be decoupled from the implementation of the server.

Implementing a new set of independent device servers meets the BattleManager's requirement for loosely coupled fully functional device servers. For these reasons, the

Polhemus and DataGlove servers were redesigned. The resulting design of the DataGlove server is described in Section 3.5, while the design of the Polhemus object is discussed in Section 3.6. The remaining objects and classes used to support the BattleManager system were either newly developed or were reused from complimentary, concurrent thesis efforts.

3.3 Structure of the BattleManager Program

The BattleManager system, in its final form as shown in Figure 3.3, makes extensive use of C++ class and object definitions developed under complementary thesis efforts. A short description of each class used in support of the BattleManager may be found in Table 3.2. For complete information concerning the TerrainGrid, GenListNode, Phigsnode, Placement, PhigsList, WorldWindow, and Translator objects see (8); for information concerning the RS-232Class, see (32).

The objects populating the synthetic environment (except for the terrain) are based on the PHIGS (Programmer's Hierarchical Interactive Graphics System) specification as implemented by the PhigsNode class. Communication between the various independent classes is coordinated by the BattleManager program. For example, the BattleManager is responsible for providing the Polhemus and DataGlove data to the FSMClass for processing. The following section describes the operation of the BattleManager system, beginning with system initialization.

3.4 BattleManager System Operation

3.4.1 System Initialization During system initialization, the synthetic object database is read and processed by an instance of the Translator class; the synthetic objects defined in the database are placed into terrain grids by an instance of the TerrainGrid class. The synthetic object database consists of three files:[5]

- The terrain database file. This file contains the polygonal description of a DMA terrain data segment, and identifies the size (width and length) of the terrain grids.

[5] For a more detailed explanation of the object database and master object files, and the processing of these files by the TerrainGrid class, see (8).

Object Name	Description
GenListNode	This object defines a single node of a generalized list. No list management functions are provided.
PhigsNode	This class defines a generalized list node used to hold PHIGS structures.
PhigsList	This class provides the list management functions needed to implement a PHIGS-like structure using the generalized list nodes defined by GenListNode.
Placement	This class provides the structure and routines to place PHIGS-based objects within the synthetic environment's coordinate system.
Socket_Class	This class implements the routines needed to create and control communications links using UNIX sockets.
DGClient	This object encapsulates the interface to the VPL DataGlove; it is a derived type of the Placement class so that the DGclient may also define the PHIGS-like DataGlove icon.
Translator	This class associates the synthetic objects by name and level of resolution to the database files which define the object.
TerrainGrid	This class defines and implements the methods needed to track and manipulate two-dimensional grids of static (non-moving) synthetic objects.
WorldWindow	This class maintains the viewing parameters, tracks the queued input devices (if any), and is responsible for rendering all terrain grids and all other PHIGS-like objects contained on its PhigsList.
Event	This object is responsible for combining the Polhemus sensor data with the data returned from the DataGlove to recognize the occurrence of defined user input events. (See Section 3.7.)
FSMClass	This class defines the finite state machine model used to convert the user gesture stream into requests for system action.
RS232Class	This class defines and implements the methods needed to control RS-232 ports.
Polhemus	This object encapsulates the interface to the Polhemus device.

Table 3.2. BattleManager Object Descriptions

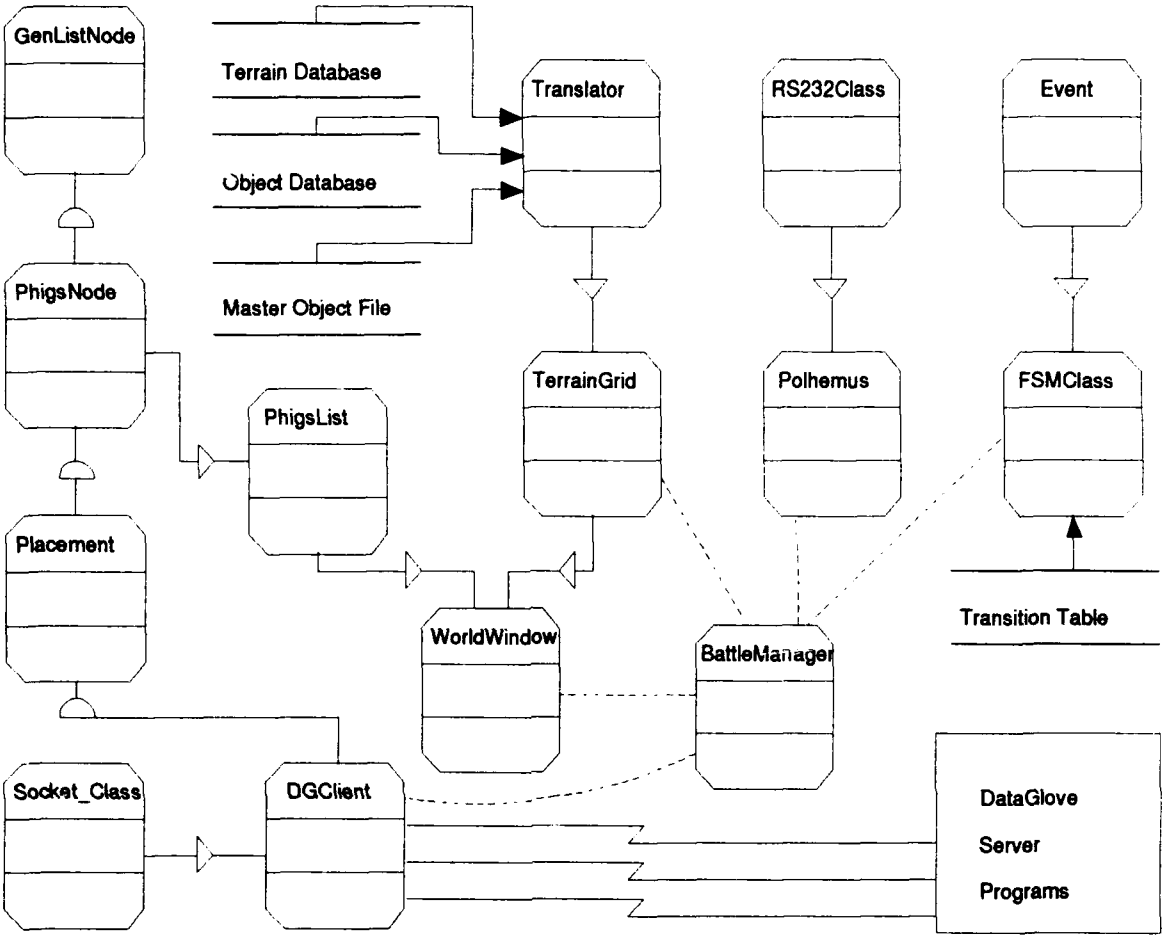


Figure 3.3. The BattleManager Synthetic Environment System.

Terrain grids are organized in a checkerboard pattern, where each terrain grid has its own enclosing bounding box used for course clipping of the terrain polygons contained in the grid. Each terrain grid also contains a list of the static (non-moving) objects located within the grid.

- The object database file. This file defines the objects that may be used to populate the synthetic environment, and includes the names of the supporting polygonal files for each synthetic object. Each synthetic object can have as many as three sets of associated polygonal files where each set of files defines the representation of the object at a given level of resolution (low, medium, and high). The intent of the multiple resolution representation is that objects in the background may be represented

using a course level of detail (few polygons), while objects in the foreground may be represented using a higher degree of detail. With multiple levels of resolution, objects that cover only a few pixels due to their distance from the eyepoint may be represented by only a few polygons, reducing the time required to render the object's image. The object database file also defines the hierarchical PHIGS-like relationship between the various components of each synthetic object (see (8)).

- The master object file. This file identifies those objects from the object database that actually populate the synthetic environment; not all objects from the object database need populate the synthetic environment. This file also identifies the world coordinate position and orientation for each object, and identifies any scaling of the object to be performed before the object is inserted into the synthetic environment. For example, if the object database contained the definition of an anti-aircraft gun, the master object file would identify the position, orientation, and scale of each anti-aircraft gun (if any) populating the synthetic environment.

Following the processing of the database files, the Polhemus object opens an RS-232 port to the Polhemus device and prepares the device for data transmission (see Section 3.6), and the DGclient object (the DataGlove client object) establishes the communication links with the remotely executing DGserver program. After the communication links are established, the DGserver affects the initialization of the DataGlove device (see Section 3.5).

Following the successful initialization of the Polhemus and the DataGlove, the DG-client object is added into the (currently empty) list of dynamic (moving) objects maintained by the WorldWindow object. The WorldWindow object is responsible for initializing and executing the rendering pipeline for the synthetic environment. For more information concerning the WorldWindow object, see (8) and (27). Following the initialization phase of the synthetic environment program, the task of rendering images and responding to user commands is begun.

3.4.2 The Main Operating Loop The main operating loop of the synthetic environment's driver program, the BattleManager, is quite simple:

```
poll the Polhemus;
poll the DataGlove:
do {
        render an image;
        read the DataGlove;
        if a full packet of data was received then poll the DataGlove;
        read the Polhemus;
        if a full packet of data was received then poll the Polhemus;
        provide the Polhemus and DataGlove data to the
        FiniteStateMachine for processing;
        obtain the system action request (if any)
        from the FiniteStateMachine object;
        perform the requested system action;
} while (system action is not terminate)
```

At system termination, the user may elect to either save the changes made to the objects in the synthetic environment, or he or she may elect to abandon all changes and restore the object database to its initial configuration. In either case, the AFIT *Flight Simulator* (see (32)) may then be executed, and user controlled simulated sorties may be flown against the objects placed into the synthetic environment by the BattleManager program.

3.4.3 Polhemus and DataGlove Control The Polhemus and DataGlove devices are operated in polled mode for three reasons:

1. In polled mode, there is no need to parse the incoming data to determine the start of a data packet; instead, a simple counter may be used to keep track of the number of bytes remaining in any given data packet; the computation time involved in reading a data packet is reduced. (For more information concerning the data formats of the DataGlove and the Polhemus, see Sections 3.5 and 3.6 respectively.)

2. Using the polled mode of operation permits a quick return from a read call if a complete data packet is unavailable; partial data packets may be assembled into a complete data packet. If the DataGlove and the Polhemus were initialized to operate in a continuous transmit mode, then a call to the read routines for these devices could not return until an entire data packet had been read, otherwise the data from several packets could be mixed together into the packet eventually returned following a successful read. The polled mode of operation helps ensure data packet integrity.
3. Using the polled mode of operation precludes the possibility of data loss due to an automatic flush of a full RS-232 port buffer.

The RS-232 ports on the Silicon Graphics 4D machines are supported by a data buffer that queues incoming data until the data buffer becomes full, at which point the buffer is flushed. The process controlling the port for the effected data buffer is not informed of the flush action. The buffer can hold only 256 characters before it is considered full (31). If a device is continually transmitting data to the RS-232 port, a low frame rate could result in data loss. Assuming each character is composed of a start bit, a stop bit, and eight data bits, and assuming a 19200 baud continuous transmission rate (as could be the case with the Polhemus), we have:

$$\frac{19200\ bits}{second} * \frac{1\ char}{10\ bits} * \frac{second}{y\ frames} = \frac{256\ char}{frame}$$

$$y = \frac{7.5\ frames}{second} \qquad (3.1)$$

If the frame rate (and hence the sampling rate) drops below 7. 5 frames per second for a continuously transmitting 19200 baud device, the data buffer for the RS-232 port to which the device is attached will be flushed and data will be lost. The polled mode of operation avoids the problems associated with the limited size of the port buffer since – at a maximum — only the number of bytes in a single data packet would be buffered at any given time.

In the paragraphs that follow, each C++ object developed in support of the whole-hand interface system is discussed, starting with the software developed and used to support the VPL DataGlove.

3.5 Software for the VPL DataGlove

Since AFIT's version of the SimGraphics software for controlling the DataGlove does not properly execute on a Silicon Graphics 4D machine, the DataGlove server incorporating the SimGraphics software is hosted on a DEC MicroVax. UNIX sockets are used to pass messages between the DataGlove server (the DGserver program) and the BattleManager's DataGlove object (the DGclient object). The relationship between the remotely hosted DGserver and the BattleManager's DGclient is based on the client-server paradigm.

3.5.1 The DataGlove Client The DataGlove client, as implemented by an instance of the DGclient class, completely encapsulates the fact that the software for controlling the DataGlove is hosted on a remote machine; the BattleManager is oblivious to the location of the executing DataGlove server. Since the DGclient must communicate with the remotely executing DataGlove server, the DGclient contains an instance of the Socket_Class. (The C++ class definitions of the DGclient and the Socket_Class may be found in Appendix B.)

The DGclient is defined to be a derived type of the Placement class for two reasons:

- The DGclient can incorporate the PHIGS-like DataGlove icon. The DGclient can then be added to the list of dynamic (moving) objects maintained by the World-Window class. (The WorldWindow class will properly render all PhigsNode and Placement class objects maintained on its PhigsList.)
- The Placement class includes the structure and methods needed to manage a list of PHIGS-like objects. These list management tools provide the capability to form a group of selected synthetic objects and perform some modeling transform such as a rotation on the entire group. No special list management tools need be developed to support the indirect modification of synthetic objects.

The DGclient defines several class methods as shown in Table 3.3. With minor exception, all method invocations are relayed to the DGserver for action. Following a command submittal, the DGclient interrogates the status connection. If no data is available from the status connection, the DGclient assumes that the requested action was successfully accomplished by the DGserver (the DGserver uses the status link only to transmit error conditions). The only DGclient method invocations that are satisfied locally are *read_dg*, *read_dg_flex*, *get_cal_file_path*, and *get_gest_file_path*. The two read methods are satisfied locally since the DGserver will transmit a DataGlove data packet to the DGclient following a poll request; upon receipt of a read request, the DGclient need only read the DataGlove data from the data socket and make the data available to the calling object.

3.5.2 The DataGlove Server The DataGlove server was initially designed to be a C++ program supported by the SimGraphics software library and by a DEC MicroVax version of the Socket_Class. However, two problems arose which resulted in a significant design change. The first problem concerned the portability of the Socket_Class, and the second problem concerned the SimGraphics shared memory segment.

3.5.2.1 Porting the Socket_Class to the DEC MicroVax The Socket_Class was initially developed and implemented on the Silicon Graphics 4D machine. Porting the software to a another platform should have been a straightforward task. However, I was unable to get the C++ version of the Socket_Class to compile on the DEC machine. Removing the C++ constructs from the Socket_Class produced the CommLink package that would successfully compile, link, and run on the DEC machine; the only difference between the Socket_Class and CommLink packages is the absence of the C++ constructs in the CommLink package. Due to time constraints, the cause of this problem was not investigated; the CommLink package was compiled to a library of routines to support the needs of the DGclient.

3.5.2.2 Development of the DGserver Program After developing the CommLink package, the C++ version of the DGserver program was completed and tested. However, the DGserver program was unable to successfully attach to the DG2P2server's shared

Method Name	Action
send_command	This private method is used to send commands to the DataGlove server.
set_cal_file_path	This method is used to set the name of the DataGlove calibration file used by the SimGraphics software.
get_cal_file_path	This method returns the name of the calibration file.
set_gest_file_path	This method sets the name of the gesture file used by the SimGraphics software in the recognition of distinguished finger postures.
get_gest_file_path	This method returns the name of the gesture file.
open_dataglove	This method accepts a defaulted parameter identifying the RS-232 port to which the DataGlove is attached, and invokes the server to open the port to the DataGlove.
calibrate	This method creates a calibration file for the current user.
init_dataglove	This method causes the initialization of the opened DataGlove. The calibration file and gesture files specified by the set_cal_file_path and set_gest_file_path respectively are loaded by the DataGlove server. Following a successful call to this method, the DataGlove is ready to report data and status.
poll_dataglove	This method requests a data packet from the DataGlove server. The method call does not wait for the receipt of a DataGlove data packet before returning.
read_dataglove	This method reads a data packet from the data communication link and returns the packet to the calling routine. If a complete datapacket is unavailable, the method will return with an error condition; depending upon the sample rate, multiple reads may be required before a complete data packet is available.
read_dataglove_flex	This method reads a modified data packet from the data communication link and returns the packet to the calling routine. The data packet read does not include the data from the DataGlove's IsoTrak sensor.
start_dataglove	This method starts the flow of data from the DataGlove device.
stop_dataglove	This method suspends output from the DataGlove. Following an invocation of this method, a new calibration and gesture file may be loaded.
close_dataglove	This method sends a request to the server to close the DataGlove port.
set_dg_read_mode	This method establishes the type of data returned by the DGserver following a poll request, and hence determines the type of data returned by the DGclient following a read request.
DG_class	This is the object's constructor. During the elaboration of the DGclient, communication with the server is established.

Table 3.3. DGclient Methods

memory block; the DGserver's requests to attach to the memory block were refused by the operating system . After some experimentation, it appeared that the ability to attach to the DG2P2server's shared memory segment was dependent upon the size of the program making the attach request. Only relatively small programs were successfully attaching to the shared memory block; larger programs were consistently denied access by the operating system to the shared memory segment. The C++ version of the DGserver program was not very large (on the order of 20,000 bytes of source code), but was much larger than the largest program granted access to the shared memory block. Due to time constraints, I was unable to determine the cause of the operating system's refusal to let larger programs attach to the DG2P2server's shared memory block (I suspect that it has to do with a size limit the operating system places on a program's data segment), but I was able to develop a work around. The C++ version of the DGserver program was split in half, with one half controlling the communications with the client, and the other half interfacing to the DG2P2server's shared memory segment. The structure of the resulting program set is shown in Figure 3.4.

3.5.2.3 The DGserver Program The DGserver program provides the client's interface to the DataGlove device. The program accepts three command line parameters identifying the network port numbers to use for the receipt of commands and transmittal of data and status. Once initiated, the DGserver program waits for a connection at the command port. Once a client connects to the command port, the DGserver waits for connections to the data and status ports. Once all three communication paths are established, the DGserver program obtains a shared memory segment for communicating with the other half of the original C++ DGserver program, the DGinterface program, and spawns the DGinterface and DG2P2server programs. It takes few seconds for the connections to be established between the various components of the DataGlove server, so the BattleManager program is designed to proceed with other processing while the DataGlove server initializes.

Since only one of the three programs comprising the DataGlove server can be executing on the CPU at any one time (the other two will be suspended), each of the three

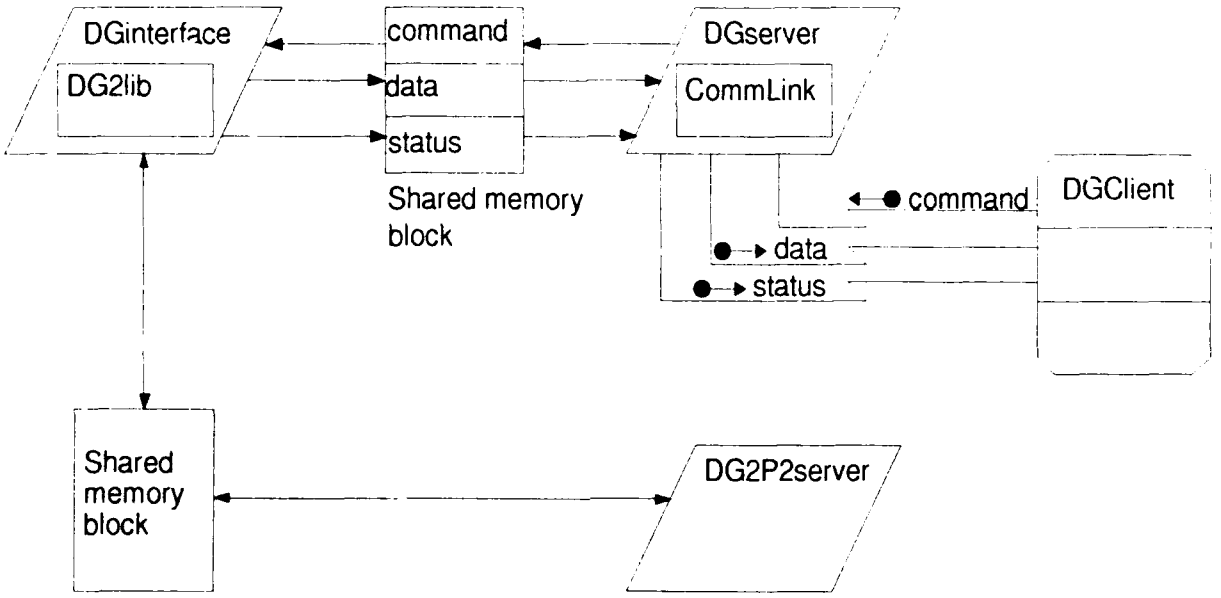


Figure 3.4. Structure of the DataGlove Server

programs was written to catch UNIX signals; only the non-suspended program will catch certain UNIX signals such as the SIGSEG signal which is generated following a segmentation fault. Any signals caught by the DGinterface or DG2P2server programs are relayed to the DGserver program. If the signal indicates that an unrecoverable error has occurred (such as a segmentation fault), the DGserver will send a user defined signal to the DGinterface and DG2P2server programs informing them to terminate. The DGserver will then notify the client of the impending server shutdown and will release its shared memory segment; the DataGlove server programs will cleanly terminate following either a system error or a user initiated termination request.

The DGserver program operates by reading the command port for incoming commands. Following the receipt of a command (from the DGclient), the DGserver writes the command into the command section of the data segment shared with the DGinterface (provided the command isn't a read command), and attempts to read the resulting data (if any) and the status of the command from the data and status sections of the shared memory segment. The shared memory segment is protected by two semaphores ensuring that the DGserver and the DGinterface programs remain "in step" with each other (*i. e.*, that neither program writes into the shared memory segment until the other program has had the opportunity to process the data that will be overwritten). Upon receipt of a read command, the DGserver sends the client (via the data link) the DataGlove data obtained from the DGinterface following the last poll request.

The format of the record sent to the client may take one of two forms depending on the type of read request received. Following the receipt of a normal read request, the DGserver will send a modified version of the data structure shown in Table 3.1. However, following a request for flex data, only the flex data, posture number, and posture name are sent to the client; this significantly reduces the number of bytes sent to the client (only about one-half as many bytes are sent as compared to a normal read). The size of this record varies, as does the size of the record depicted in Table 3.1. To prevent the need for the client to parse the incoming data to determine the end of the DataGlove data record, each record is prefixed with a short (a 16 bit integer) identifying the length of the remaining data record. A carriage return is appended after the length field to separate it from the rest of the record. The data records sent by the DGserver thus have the format shown in Table 3.4; the entries in square brackets are optional depending on the type of read request received.

length	(return)	[position]	[orientation]	flex data	posture #	posture name
1–2 bytes	1 byte	21 bytes	21 bytes	2–3 bytes each	1–2 bytes	upto 16 bytes

Table 3.4. DGserver Data Record Format

3.5.2.4 The DGinterface Program The DGinterface program forms the interface between the DGserver program and the DG2lib routines. The DGinterface operates

by reading command requests from the command section of the memory segment shared with the DGserver. The command requests are translated into calls to the appropriate DG2lib routines, and the resulting status and data (if any) are written into the DGserver's shared memory segment.

Since the BattleManager is supported by a Polhemus 3-Space Tracker, the DataGlove's IsoTrak is not needed; one of the sensors from the Polhemus is used to track the DataGlove. For this reason, the BattleManager uses the *read_dg_flex* method to obtain the DataGlove data.

3.6 The Polhemus 3-Space Tracker

The Polhemus 3-Space Tracker uses "low-frequency, magnetic field technology to determine the position and orientation of a sensor in relation to a source or other specified reference frame — providing a full six-degree-of-freedom measuring device. " (28:1-1). The source is fixed in a central location in the area of intended use. In addition to the source, the Polhemus is composed of one to eight sensors and a systems electronic unit (SEU).

Communication between SEU and the host can take place either through either an RS-232 or an RS-488 connection. The RS-232 baud rate is user selectable; baud rates of 300 to 19200 are supported.

The data returned from the SEU following a read request is under user control. The data items that can be returned include cartesian coordinates, orientation angles, quaternions, and the X, Y, and Z axis direction cosines. The Polhemus is completely programmable; the user may define both the data types to be returned (quaternions, position, *etc.*) and the ordering of the data in the data packet.

As shown in Table 3.5, the Polhemus class is designed to support three basic data packet structures: the *orientation_angle_packet*, *direction_cosine_packet*, and the *quaternion_packet*. The BattleManager — via an instance of the Polhemus class — initializes the Polhemus so that the data returned includes direction cosines since they can be used directly to define a rotation and translation matrix. In the case of the Silicon Graphics

```
typedef struct {
                float                   x, y, z;            /* direction cosine */
}               direction_cosine_type;

typedef struct {
                float                   q0, q1, q2, q3;     /* quaternions */
}               quaternion_type;

typedef struct {
                float                   x, y, z;            /* cartesian coordinates */
}               position_type;

typedef struct {
                position_type           position;
                float                   yaw, pitch, roll;   /* orientation angles */
}               orientation_angle_packet;

typedef struct {
                position_type           position;
                direction_cosine_type   DX;                 /* X-axis direction cosine */
                direction_cosine_type   DY;                 /* Y-axis direction cosine */
                direction_cosine_type   DZ;                 /* Z-axis direction cosine */
}               direction_cosine_packet;

typedef struct {
                position_type           position;
                quaternion_type         quaternion;
}               quaternion_packet;
```

Table 3.5. Polhemus_Class Record Formats

4D, the matrix has the form:

$$\begin{pmatrix} [X - axis\ direction\ cosine] & 0 \\ [Y - axis\ direction\ cosine] & 0 \\ [Z - axis\ direction\ cosine] & 0 \\ [position] & 1 \end{pmatrix}$$

The *position* entry in the above matrix is adjusted from the Polhemus' reference frame to the synthetic environment's reference frame. The readings from one of the two sensors is used as a rotation and translation matrix for the DataGlove icon, while the readings from

the other sensor are used to define the viewing parameters.

The BattleManager's view volume is defined using the view reference point (VRP), view up vector ($V\vec{U}P$), view plane normal ($V\vec{P}N$), and perspective reference point (PRP) as described in (17). The view volume is designed to take advantage of the direction cosine data. Specifically, $V\vec{U}P$ is defined by the Z-axis direction cosine, the $V\vec{P}N$ is defined by the negative of the X-axis direction cosine, and the VRP is defined by the Polhemus station's position value corrected to the synthetic environment's coordinate system. Since the BattleManager allows the user to manipulate the viewpoint using gesture streams (see Section 3.7), an offset is added to the adjusted position value so that the viewpoint isn't tied to the physical location of the user. To combat the effects of noise (see Section 2.2.2), the Polhemus' position information is run through a simple filter to reduce the effects of noise and jitter: positional information that varies less than a threshold is ignored, and the position of the DataGlove icon (or the viewpoint as the case may be) remains unchanged. The direction cosines and positional information from the sensor attached to the DataGlove is used in the recognition of whole-hand gestures.

3.7 The Finite State Machine

The Finite State Machine implemented as part of the whole-hand interface provides the user a great deal of latitude for tailoring the synthetic environment's interface. Each user may associate a different gesture or gesture stream with a given system action without the need to recompile the software; the interpretation of the gesture stream is completely user definable. It is the finite state machine model that defines the relationship between system actions and the gestures formed by the user.

3.7.1 Definition of the Finite State Model Finite state machines can be defined as a sextuple (12:191):

$$(Q, q_0, \Sigma, \Phi, f, g)$$

where

$$
\begin{aligned}
Q &= \textit{nonempty finite set of states} \\
q_0 &\in Q \textit{ is called the initial state} \\
\Sigma &= \textit{nonempty set of input symbols} \\
\Phi &= \textit{nonempty finite set of output symbols} \\
f &= \textit{state transition function}, \\
&\quad f: Q \times \Sigma \rightarrow Q \\
g &= \textit{output function}, \\
&\quad g: Q \times \Sigma \rightarrow \Phi
\end{aligned}
$$

The f function defines, for a given state, the next state associated with an input symbol σ, while the g function defines the output symbol ϕ generated during the transition. For example, if $f(q,a) = q'$ and $g(q,a) = b$, then from the state q the finite state machine would transition to state q' on receipt of the a input symbol and would generate the output symbol b during the transition. The finite state machine implemented as part of the whole-hand interface system allows the user to define both the f and g functions. In the BattleManager system, the input alphabet Σ is the set of recognizable gestures, and the set of output symbols Φ is defined to be the set of synthetic environment actions, such as moving an object. The sets Σ and Φ are discussed in Sections 3.7.2 and 3.7.3 respectively.

Using the finite state machine model for interpreting the gesture stream affords the user complete control over the actions associated with each gesture based on the system context. For example, the THUMBS_UP input symbol (an oriented posture) may be associated with an affirmative acknowledgement in one case or with a request to save the last modifications made to a group of synthetic objects in another case. The system action requested would be based on the definition of the g function which in turn depends upon the current state of the finite state machine.

In the BattleManager system, the finite state machine is defined by a modified state transition table; the transition table defines both the f and g functions. The finite state

machine implemented to support the BattleManager does not require defined transition for all input symbols for a given state; if an input symbol σ is received with the finite state machine in a state q without a defined transition for the symbol σ from state q, the finite state machine will ignore the input symbol. A sample transition table is shown in Table 3.6. (The input and output symbols used in the table are defined in Sections 3.7.2 and 3.7.3 respectively.) This transition table produces the finite state machine depicted in Figure 3.5. To change the gesture (or gestures) responsible for affecting a system action requires modification only to the state transition table; the software remains unchanged.

Current State	Input Symbol	Output Symbol	Next State
0	Horns	Hilite_First	15
15	Null	Ready	16
16	Thumbs_Down	Ready	18
16	Thumbs_Up	Group	18
16	Horns	Rotate_Start	17
18	Null	Hilite_Next	16
17	Null	Rotate_Stop	19
19	Horns	Rotate_Start	17
19	Thumbs_Up	Done_Save	20
19	Thumbs_Down	Done_Abandon	20
20	Null	Ready	0

Table 3.6. Sample Transition Table

The FSMClass defines the finite state model used in support of the whole-hand interface. As shown in Table 3.7, the class supports three public methods in addition to the destructor. Additional private methods are used during the processing of the transition table. During the elaboration of an FSMClass object, the transition table is prccessed and the finite state machine is defined. Input symbols for the finite state machine are generated from DataGlove and Polhemus data by an included Event object (see section 3.7.3). Once the finite state machine is defined, processing of the gesture stream may begin.

3.7.2 Input Symbol Recognition The included Event class performs the task of combining the posture information from the DataGlove with the position and orientation information from the Polhemus; the Event class encapsulates the device dependencies associated with recognizing the occurrence of user input events. The input events are translated by

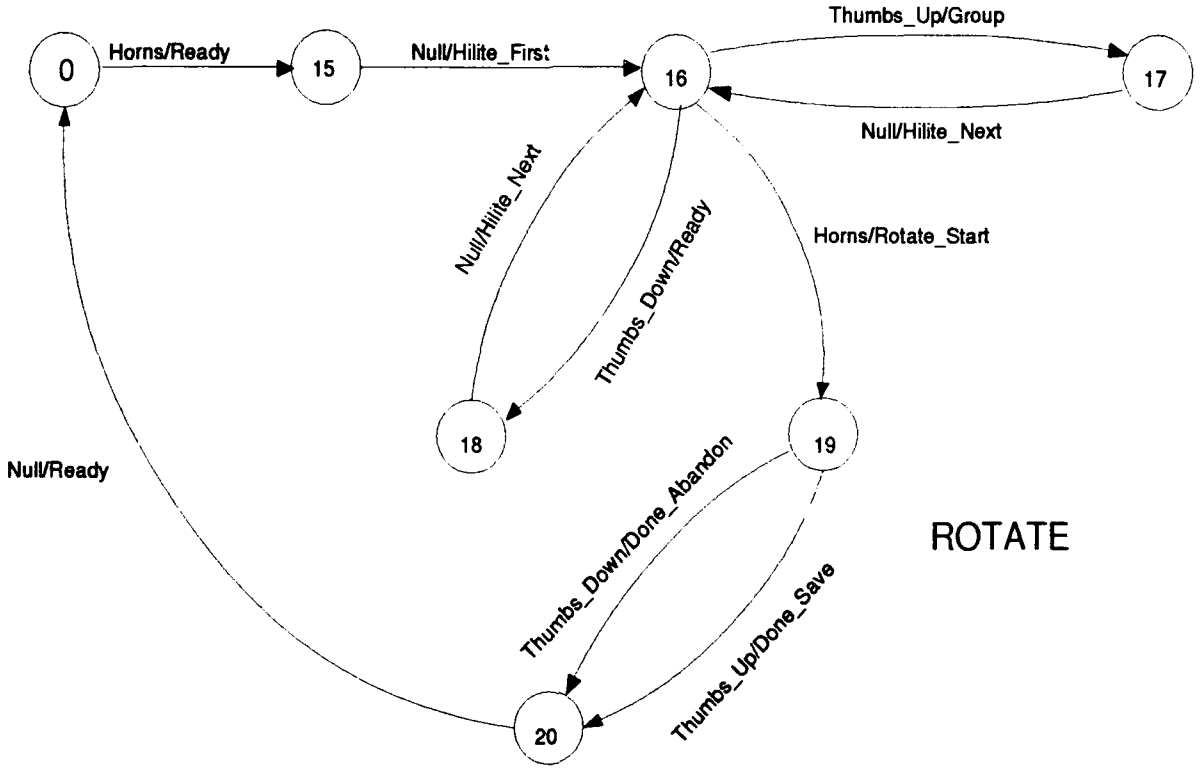


Figure 3.5. Sample State Transition Diagram

the Event class into input symbols for the FSMClass object. Some input events, such as SAFE, require a comparison of either position or orientation data from a given data sample with a baselined data sample. For example, the SAFE input event is generated by sweeping a palm posture (a flat hand) through a specified distance. To recognize the SAFE event, the position of the hand must be recorded at the first occurrence of the palm posture whenever the SAFE symbol[6] is used to define a transition from the current state of the finite state machine. Subsequent positional information may then be used to determine the occurrence of the SAFE gesture. If the palm posture is lost during the sweep (*i. e.*, the user changes his or her finger posture while translating the hand), the baseline value must

[6]Note the similarity between symbols and events. The occurrence of an input event will result in the generation of an input symbol.

Method Name	Action
define_transition	This private method is used to define the finite state machine transition function (f) and output function (g).
update_state	This method accepts the DataGlove and Polhemus data, updates the current state of the finite state machine, and returns the system action defined by the transition.
reset	This method resets the finite state machine to the initial state. The system action returned is *Ready*.

Table 3.7. FSMClass Methods

be re-established. To aid in the recognition of these gestures, the Event class includes the method *begin_watch_for_event (event)* where *event* is gesture whose baseline data needs to be established. Upon entering a new state, the FSMClass processes the list of defined transitions out of the current state and calls the *begin_watch_for_event (event)* method if needed. The set of input symbols, Σ, is defined in Table 3.8.

3.7.3 Output Symbol Definitions The output symbols of the finite state machine are used to request specific synthetic environment actions; the output symbols have a one to one correspondence with the defined system actions. System actions are defined to support indirect manipulation of the synthetic environment objects. [7] The following steps must be taken in order to manipulate the synthetic objects:

1. An object must be highlighted. Highlighting an object extracts from the Terrain-Grid the information needed to identify the object to the BattleManager program. The highlighted object will be rendered solid red to distinguish it from other non-highlighted objects. To aid the user in locating the highlighted object, a line from the DataGlove icon will extend to the highlighted object. Figure 3.6 shows a highlight procedure in action. Objects may be highlighted in succession until the object of interest is found.

[7] The user of the synthetic environment manipulates the environment's objects through gesturing. The user does not need to — and in fact cannot — physically position the DataGlove icon to pick-up and manipulate the objects directly.

Input Symbol	Type	How Formed
Horns	Finger Posture	Pinky and index finger extended, other fingers drawn towards the palm.
Ok	Finger Posture	Pinky, ring, and middle finger extended, thumb and index finger drawn towards the palm.
Fist	Finger Posture	Self explanatory.
Index	Finger Posture	Only the index finger extended.
IndexMiddle	Finger Posture	Same as the Index posture, except that the middle finger is also extended.
Palm	Finger Posture	All fingers extended.
Scout	Finger Posture	The boyscout salute: the index, middle, and ring fingers extended, the other fingers drawn toward to palm.
Thumbs_Up	Oriented Posture	Self explanatory.
Thumbs_Down	Oriented Posture	Self explanatory.
Stop	Oriented Posture	Palm posture where the hand is oriented with the fingers up, similar to a traffic policeman's stop signal.
Cut	Moving Posture	The inner knuckles of the pinky, ring, middle, and index fingers bent 90 degrees, followed by a wrist rotation of at least 60 degrees.
Safe	Moving Posture	Palm posture swept through some minimum distance; similar to the safe signal used in baseball.
Wrist_Twist	Hand Gesture	A twist of the wrist of at least 60 degrees about an imaginary axis running parallel through the forearm.
Null	Don't Care	Any posture other than those listed above.

Table 3.8. Input Symbols

2. An object must be selected. This action extracts the highlighted object from the TerrainGrid and attaches it to the PhigsList contained within the DGclient. Once the object is selected, it is rendered solid green, giving a visual indication that the user may proceed with the transformation. Multiple objects may be selected by repeating steps one and two as needed.

3. A modeling transformation is applied, such as a rotation or scale. The transformation selected will be applied to all objects on the DGclient's PhigsList. Multiple transformations may be applied in succession.

4. The changes should either be saved or abandoned. Saving the changes detaches the objects from the DGclient's PhigsList and — with the exception of *delete* — places them back into the TerrainGrid. Abandoning the changes will also detach the objects

from the DGclient's PhigsList, except that the objects are restored to their original condition (position, orientation, and scale) before they are inserted back into the TerrainGrid.

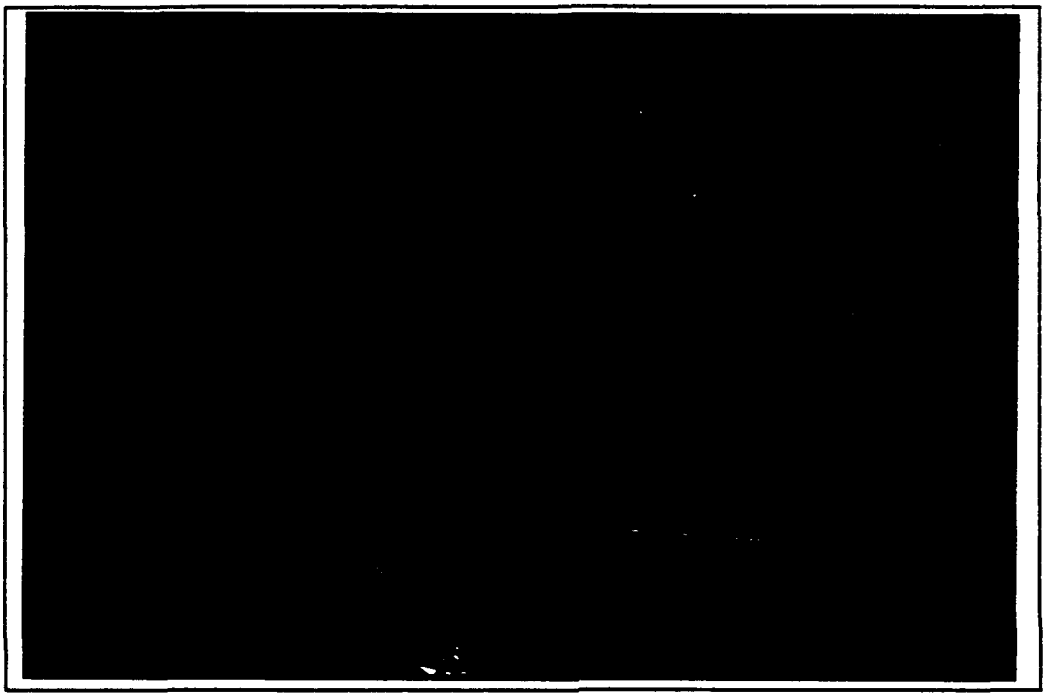


Figure 3.6. Highlighting a Synthetic Object.

The available system actions include the following:

- *Ready* No system action taken.
- *Hilite_First* This action results in highlighting the synthetic object nearest the DataGlove icon. The object is colored red to indicate the highlight, and a line is drawn from the DataGlove icon to the object to aid the user in locating the object. (The nearest object could be out of view, hidden by either terrain or clipped out of the viewing volume due to the location of the object.).
- *Hilite_Next* This requests that the next nearest object be highlighted.
- *Group* This action removes the highlighted object (if any) from its terrain grid and adds the it to the DGclient's PhigsList structure for later manipulation. The objects

do not alter their position or orientation, but are colored red as an indication of their grouping.

- *Copy* This action duplicates all grouped objects. The duplicated objects remain attached to the DataGlove, while the original group of objects are reinserted into the terrain grids. The copied objects retain their red coloring until they are detached from the DataGlove.
- *Delete* All grouped objects are deleted.
- *Translate_Begin* All grouped objects become slaved to the position of the DataGlove icon; after invoking this system action, the grouped objects will be translated as the DataGlove icon is moved. The relative positions between the objects will remain unchanged. For example, if the grouping consists of three anti-aircraft guns, the relative position and orientation of the guns with respect to each other will be preserved; only the position of the group will be altered.
- *Translate_End* Invoking this action decouples the position of the grouped objects from the position of the DataGlove; the objects remain attached to the DataGlove, and hence retain their highlighted color.
- *Rotate_Begin* This action is similar to Translate_Begin, except that it is the orientation of the grouped objects that is slaved to the orientation of the DataGlove. If several objects are grouped, the objects will be rotated about the center of the group.
- *Rotate_End* This action decouples the orientation of the grouped objects from the orientation of the DataGlove. The objects remain attached to the DataGlove, and therefore remain highlighted.
- *Scale_Begin* Invoking this method records the position of the DataGlove icon, and begins to scale the grouped objects based on the relative displacement of the DataGlove from its baselined position. For example, if after beginning to scale, the DataGlove is moved three units up the Z-axis, the grouped objects will increase 30% in size. The scaling factor is a constant of one (scaled) Polhemus unit to a ten percent change in scale, with a minimum scale change of one percent.

- *Scale_End* This action terminates the scaling effect caused by translating the DataGlove. The objects remain attached to the DataGlove and therefore remain highlighted.
- *Move_DOV* Invoking this action results in moving the eyepoint one unit along the direction of view (DOV). Since the DataGlove's position is relative to that of the eyepoint, DataGlove will keep its relative position with respect to the eyepoint.
- *Move_Straight* Invoking this action causes the eyepoint to be moved one unit in the direction parallel to the sum of the X and Y components of the DOV; the height of the viewpoint will remain unchanged. As with the Move_DOV action, the DataGlove will track with the eyepoint.
- *Speed* This action increases the *Move_Straight* and *Move_DOV* speed by approximately ten percent.
- *Slow* This action decreases the move speed by approximately ten percent.
- *Continuous_Move* This action causes the system to continuously move the eyepoint (and hence the absolute position of the DataGlove icon) in the direction determined by the last move command.
- *Stop_Continuous_Move* This action terminates the Continuous_Move action.
- *Rev_Dir* This action reverses the move direction.
- *Raise_Eyepoint* This action raises the eyepoint.
- *Lower_Eyepoint* This action lowers the eyepoint.
- *Raise_Group* This action increases the vertical displacement of the grouped objects.
- *Lower_Group* This action decreases the vertical displacement of the grouped objects.
- *Done_Save* This action saves the results of the modeling transforms that were applied to the grouped objects, and detaches these objects from the DataGlove. The objects return to their normal coloration following execution of this action.
- *Done_Abandon* This action abandons the effects of the modeling transforms applied to the grouped actions (including the Copy and Delete actions). All grouped objects are

returned to their original position, orientation, and scale. The DGclient's PhigsList is also cleared.

- *Exit_Save* Invoking this system action updates the synthetic object database files and exits the system.
- *Exit_Abandon* This system action simply exists the system; any changes made to the objects in the synthetic environment are lost.

3.8 Summary

This chapter discussed the constraints placed on the design of the BattleManager system, and described the use of the state based model for interpreting the gesture stream.

As shown in Figure 3.6, the DataGlove icon can be colored to correspond with the current state of the state based model. For example, if the user develops a finite state machine for duplicating a group of synthetic objects, the DataGlove icon could be rendered in a color identifying to the user the current state of the duplication effort. The visual cue provided by the color of the DataGlove icon could help prevent inadvertent application of the modeling transforms.[8] The following chapter contains a series of color plates showing the manipulation of a synthetic object using the whole-hand interface system.

[8]Changing the color of the DataGlove icon is similar in intent to the way computer-based drawing packages like MicroSoft's DrawPerfect change the representation of the mouse cursor to reflect the selected editing action.

IV. Results and Recommendations

As mentioned earlier, the goal of this effort was to develop a state based whole-hand interface to a synthetic environment. The interface developed and implemented in the BattleManager program allows a user to associate context sensitive DataGlove gestures with requests for system action. The state based system was developed prior to the discovery of the paper by Green (19). Green's paper discusses three methods for defining user interfaces, including the state based and the event based methods. The BattleManager system combines some of the techniques discussed by Green with other synthetic environment interface techniques.

The visual cues provided by altering the color of both the DataGlove and selected synthetic objects based on the actions performed by the user enhance the system interface. Combining the Polhemus data with the finger posture data resulted in a much richer set of user commands than could be realized from the posture data alone. The combination of the state based model with the whole-hand interface paradigm provides a very powerful interface technique. The following section demonstrates the effectiveness of both the visual cues and the state based model.

4.1 Effectiveness of the Finite State Model

As mentioned earlier, a variety of modeling transforms may be applied to the synthetic objects from within the synthetic environment. Figures 4.1 through 4.5 show a user manipulating a synthetic object. In Figure 4.1, the user has located the object of interest, and is preparing to manipulate it. In Figure 4.2, the user has formed the gesture he or she has associated with highlighting an object. The object's color has changed to indicate that it has been highlighted, and the color of the DataGlove icon has been changed to indicate the action which will be applied to the object. In Figure 4.3, the user has selected (grouped) the aircraft. The color of the aircraft has again changed to inform the user that the system is ready to manipulate the object. Both aircraft were ultimately grouped. In Figure 4.4, the user has rotated the group. The aircraft remain highlighted since they have not yet be inserted into the terrain grid. In Figure 4.5, the user has inserted the aircraft

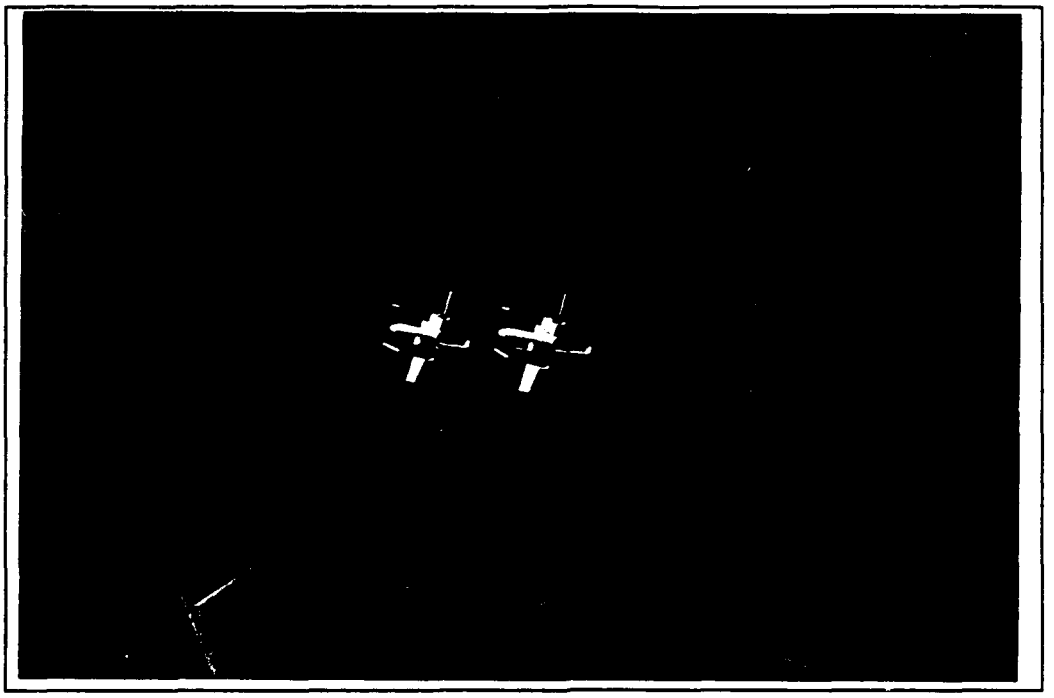


Figure 4.1. The user has located the synthetic objects.

into the terrain grid. The DataGlove icon and the aircraft return to their normal color, providing a visual cue to the user that the requested action is completed.

The finite state machine used to perform the above action was defined by the user of the system. Each user may define their own finite state machine model for interacting with the synthetic environment. For example, I prefer to use the THUMBS_UP oriented posture to request the *group* system action after highlighting an object. Another user might want to use a different gesture (or gesture stream) for requesting the *group* action; the finite state machine can be as simple or as complex as the user desires. To alter the operation of the system, the user need only modify the transition table defining the finite state machine.

Although the state based model works well, no software project is ever complete. The BattleManager program is no exception to this rule. The following section addresses some of the limitations of the state based model.

Figure 4.2. The object of interest has been highlighted. Note the color change.

4.1.1 System Limitations The state based model as implemented in the BattleManager program does not check the integrity of the state transition table. The system does not check for states which are either sources or sinks, [1] nor does the system check to ensure that the resulting finite state machine is connected; it is possible to define a group of states that cannot be reached from the initial state.

The set of system actions, Φ, is by no means complete. Only a minimal set of operations are currently supported. To be fully functional, additional system actions such as a request to find a misplaced (lost) object need to be added. There is no defined system action for taking an object from the object database file (see Section 3.4) and inserting into the synthetic environment; only those objects initially placed into the environment via the master object file are available for manipulation.

Only one input event per frame is recognized by the current system. If multiple users were exploring the same synthetic environment, multiple input events could occur in every

[1] A source state is one with only outbound transitions; there is no way to enter the state unless the system starts in that state. A sink state is one with only inbound transitions. Once the state is entered, there is no way to leave the state.

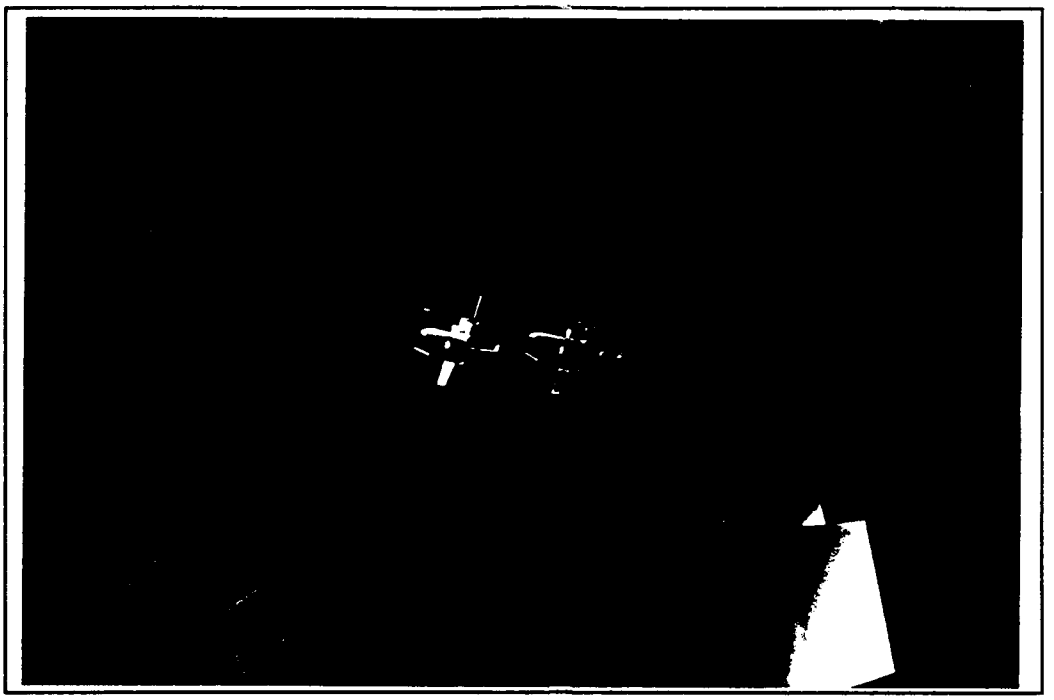


Figure 4.3. The object has been grouped. Note the color change.

frame. The Event class could be modified to support the collection and ordering of input events obtained both from the local user and from the networked users. Each user could define their own finite state machine for interacting with the synthetic environment, while a separate finite state machine common to each remote site could be used to interpret the network activity.

4.1.2 Other Application Areas There is no reason why the finite state model cannot be used to interpret input symbols from other input devices. The Event class could be generalized to support a wide range of interface devices. Each input device could define an instance of the Event class, and using the inheritance offered by C++, instantiate an Event object which would recognize the occurrence of input events specific to the device. These input events could by translated into input symbols for the finite state machine, where they in turn would be translated into requests for system action.

The finite state machine could be modified to support nondeterminism. The nondeterministic model could be used in gaming. For example, a state based synthetic environment for war-gaming could be developed. The nondeterministic nature of the finite state

Figure 4.4. The aircraft have been rotated.

machine could be used to enhance the reality of the simulation: Suppose an aircraft commander formed the gesture (or gesture stream) for strafing an enemy convoy. The finite state machine could support multiple next states for that action, ranging from completely missing the convoy and getting shot down to destroying several key vehicles. To alter the possible outcomes requires only a change to the definition of the transition table for the finite state machine, and should not require a change to the underlying software.

4.2 Further Research and Development

One of the more distracting features of the BattleManager system is due to noise and jitter: the position and orientation of the DataGlove icon and the view controlled by the Polhemus device are influenced by perturbations of the Polhemus' magnetic field. The Polhemus data could be filtered in an effort to reduce the effects of noise. If the Polhemus device was remotely hosted, the remote host potentially could filter the data without reducing the frame rate of the system since the filtering task would be off loaded from the rendering engine. The unfiltered position and orientation data obtained from the

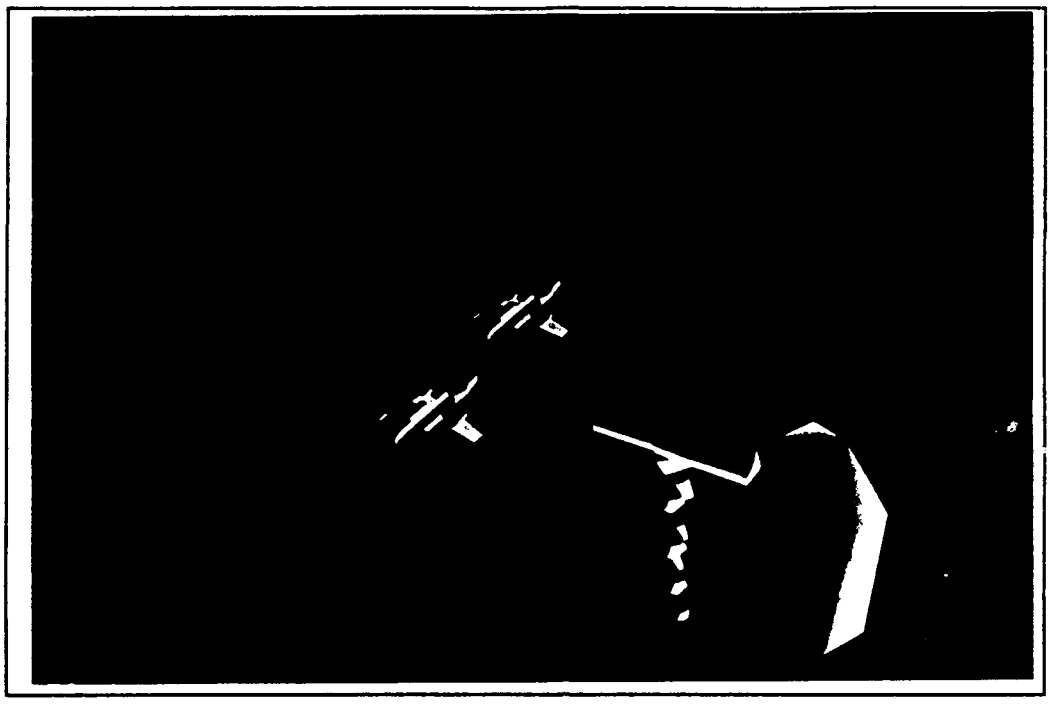


Figure 4.5. The rotated objects have been inserted into the terrain grid.

Polhemus is not nearly as accurate or stable as the data that could be obtained from a mouse or similar input device.

4.3 Conclusion

The state based interface model has been successful in providing the capability to tailor one aspect of the user interface to a synthetic environment: the interpretation of the gesture stream. The state based approach provides the user the flexibility to alter the interpretation of user input without the need to alter the underlying software.

Appendix A. *Domain Analysis*

Before the BattleManager system was constructed, I performed a limited Domain Analysis of the synthetic environment system. Part of this analysis involved research into the tools and techniques for interacting with synthetic environments; the results of this research are contained in Chapter 2. Another part of this analysis consisted of identifying the operations, objects, and structures of a synthetic environment. Of the three Domain Analysis tools (Concept Maps, Event-Response Lists, and Storyboards), only Concept Maps were used. The Concept Maps I developed early in the development of the BattleManager system are presented below. Due to time constraints, not all of the relationships depicted on the Concept Maps were implemented. For instance, the *Threat* object of Figure A.4 was not implemented.

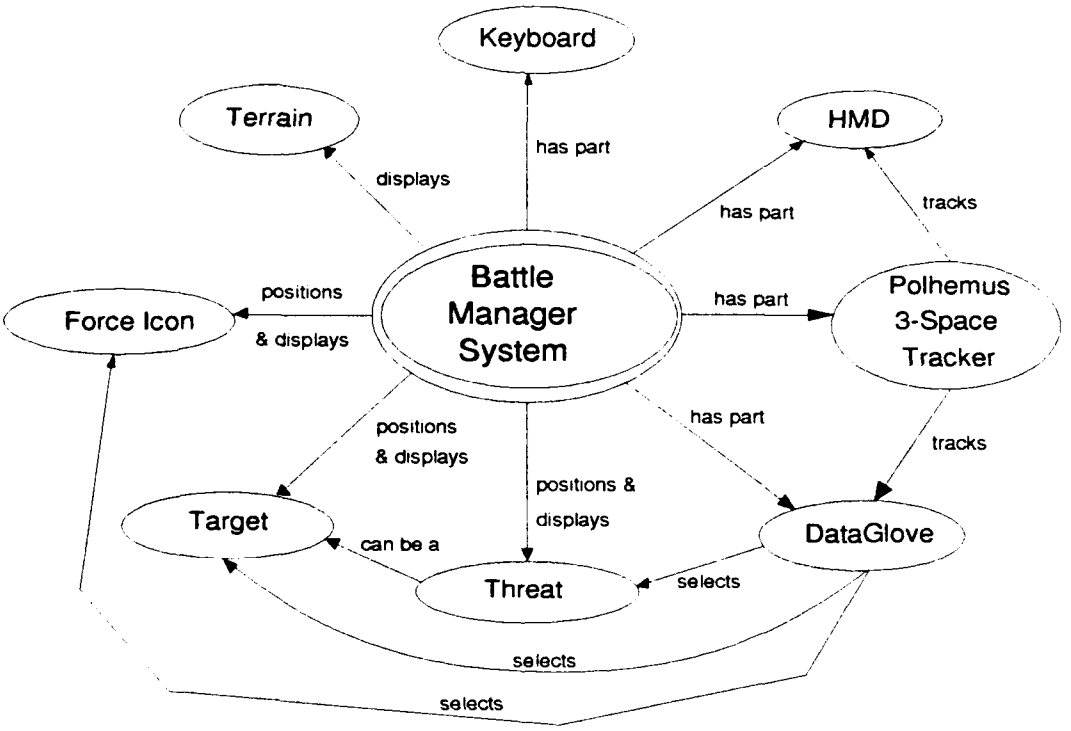


Figure A.1. BattleManager Concept Map

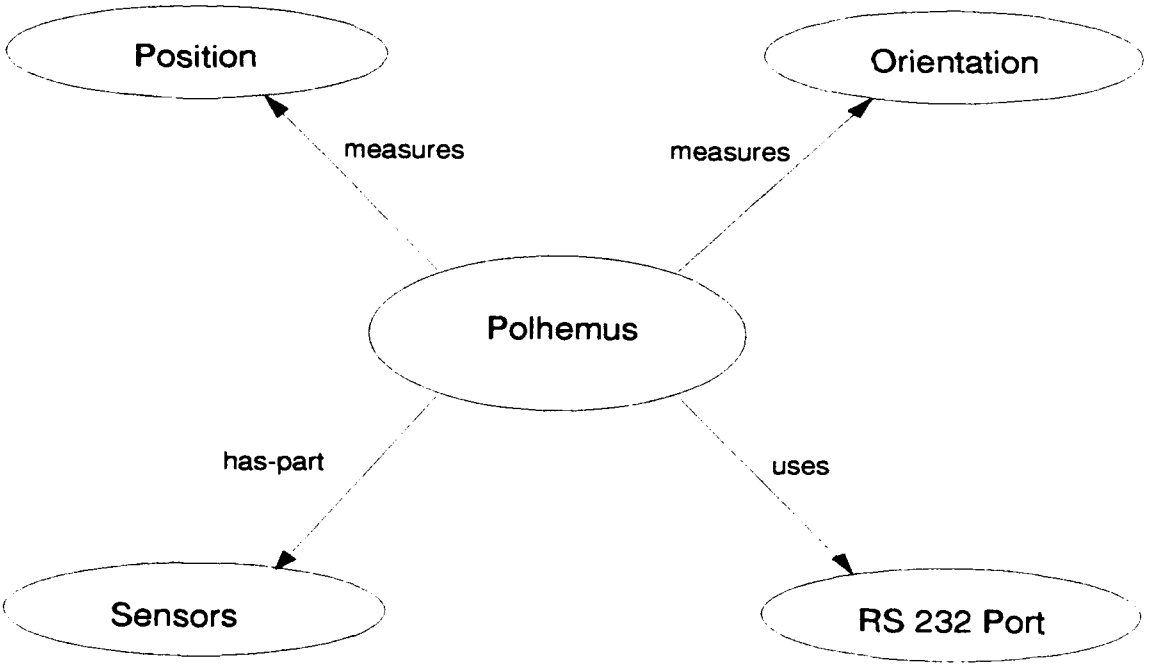


Figure A.2. Polhemus Concept Map

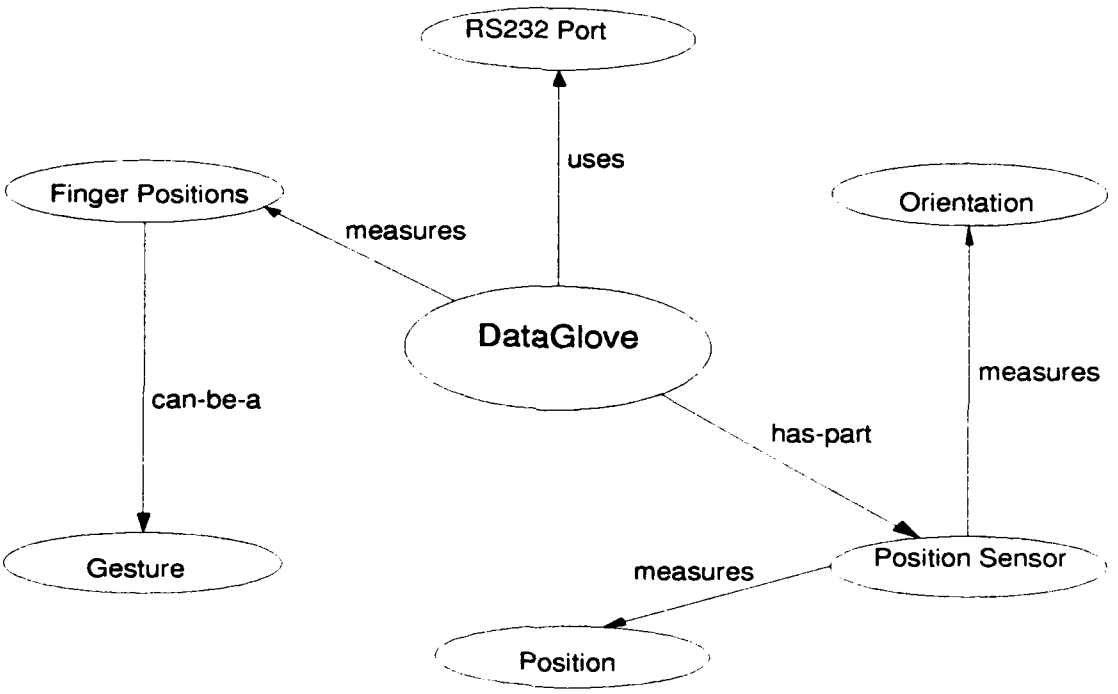


Figure A.3. DataGlove Concept Map

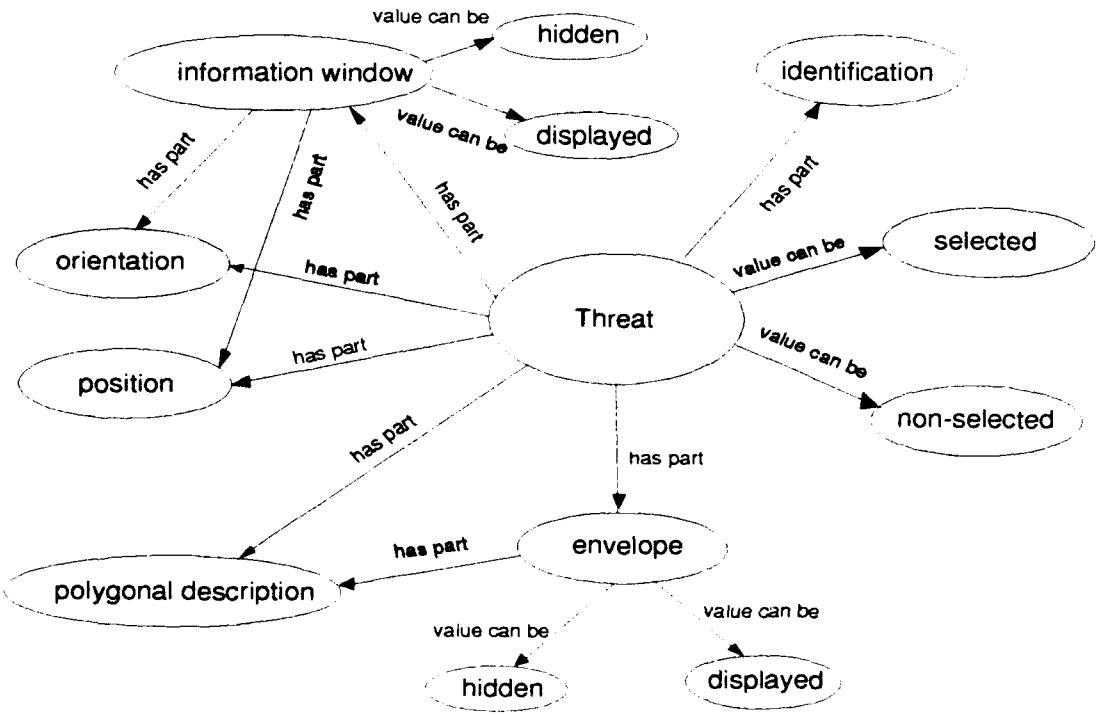


Figure A.4. Threat Concept Map

Appendix B. *C++ Class Definitions*

This appendix contains the C++ header files defining the Polhemus, DGclient, Socket_Class, FSMClass, and Event classes. For definitins of the GenListNode, PhigsNode, Placement, PhigList, Translator, TerrainGrid, and WorldWindow classes, see (8). For the definition of the RS232Class, see (32).

```
// ------------------------------------------------------------------------
// DATE   :  18 Sep 91
// VERSION:  1.8
//
// FILE NAME   :  Polhemus.h
// DESCRIPTION :  This is the header file for the C++ class object for the
//    Polhemus 3-Space Tracker.  Since the methods exported by this class
//    object use the RS-232 ports, the Polhemus class contains a
//    Managed_RS232_Port class object for controlling the serial port.
//
//    NOTE:  Whenever a set_Polhemus method is invoked, the calling object
//    is responsible for waiting a sufficiently long period of time (<1 sec)
//    for the set_Polhemus command to take effect.  For example, following
//    a call to set_Polhemus_boresight, the calling object should proceed
//    with other processing (or delay) before attempting to read data from
//    the Polhemus.  A boolean flag, Wait, is included in the parameter list
//    for this purpose:  If true, the method waits 1 second before returning
//    control back to the calling method.
//
//
// EXPORTED METHODS:
//    Polhemus_Class () -- Constructor.  Sets the default values for
//                         the operational envelope, the source height,
//                         the read mode, and the Read_in_progress flag.
//                         Invokes the initialize_Polhemus method.
//    initialize_Polhemus() -- This method initializes the Polhemus
//                         device and prepares it for operation.  Note that
//                         this method takes several seconds to complete.
//    poll_Polhemus () --  This method sends a command to the Polhemus
//                         telling it to report the status of the sensors.
//                         Following a poll command, the Polhemus sensor
//                         data may be read.
//    read_Polhemus    --  This method is constructed so that either
//                         Quaternions, Orientation, or Direction Cosines
//                         may be read from the Polhemus device.  The method
//                         is overloaded to achieve this capability.  Note
//                         that the Polhemus must be properly initialized
//                         before any reads take place.
//                         The status byte from the Polhemus is returned
//                         with a successful read attempt.  Note that this
//                         status byte should be inspected by the calling
//                         object since, depending on the status byte, the
//                         data returned by the read might be invalid.
//    read_Adjusted_Polhemus -- This method performs the same function
//                         as read_Polhemus except that the sensor values
//                         are adjusted for the lab's environment:  The Y
//                         and Z axis values of the position are negated,
//                         and the Z axis value is adjusted for the source
//                         height.  Corresponding corrections are made for
//                         the direction cosine values. Note that quaternion
//                         and orientation values remain unchanged.
```

```
//                          The status byte from the Polhemus is returned
//                          with a successful read attempt.  Note that this
//                          status byte should be inspected by the calling
//                          object since, depending on the status byte, the
//                          data returned by the read might be invalid.
//    read_Adjusted_Polhemus -- This method performs the same function
//    set_Polhemus_height -- This method sets the attribute for the height
//                          of the Polhemus source.
//    set_Polhemus_read_mode -- This method sets the attribute defining the
//                          type of information returned by a read request.
//    set_Polhemus_envelope -- This method sets the operational envelope of
//                          a station (source-sensor pair).
//    set_Polhemus_hemisphere -- This method sets the Hemisphere attribute
//                          and uses it to define the operational hemisphere
//                          of a station.
//    activate_Polhemus_station -- Activates a station (sensor).  Note that
//                          this method takes a few seconds to complete.
//    deactivate_Polhemus_station -- Deactivates a station.  Note that this
//                          method takes a few seconds to complete.
//    set_Polhemus_to_inches -- This method sets the Polhemus device to
//                          return position information in inches.
//    set_Polhemus_to_centimeters -- This method sets the Polhemus device to
//                          return position information for a given station
//                          in centimeters.  Note that the default
//                          Polhemus_height is in inches.
//    set_Polhemus_boresight -- Sets the default orientation for a station.
//                          Note that when this method is invoked,
//                          all subsequent orientation information
//                          retrieved from the Polhemus is relative to the
//                          orientation of the sensor at the time the
//                          set_Polhemus_boresight command was invoked.
//    reset_Polhemus_boresight -- This method resets the Polhemus boresight
//                          to the device's default value.
//    define_Allignment -- This method establishes the allignment reference
//                          frame for the Polhemus.
//    reset_Allignment  -- This method resets the allignment to the default
//                          allignment.
//    reset_Polhemus --     This method resets the Polhemus.  Note that this
//                          method takes several seconds to complete.
//    set_scale_factor --   This method is used to set the ratio of the scale
//                          used by the Polhemus (in other words, this method
//                          scales the Polhemus position values by the
//                          scale factor.  For instance, if 1 inch absolute
//                          distance corresponds to 10 feet in the model,
//                          then the scale_factor should be set to 1:120,
//                          or 0.0083.)
// ------------------------------------------------------------------------

#ifndef _POLHEMUS_H_
#define _POLHEMUS_H_
```

```
#include "globals.h"
#include <stdio.h>

#include "uRS232port.h"


// ------------------------------------------------------------------------
// --------------          Class Definition          --------------------
// ------------------------------------------------------------------------


class Polhemus_Class  {


   public:       // typedefs, #defines, and structures needed by
                 // the class follow

      // ------------------------------------------------------------------
      // Define the record size returned from each station for a given poll
      // request.
      //   3 bytes overhead
      //  21 bytes position information
      //   1 byte space  PLUS
      //
      //  for cosines:
      //     21 bytes X direction cosine
      //      1 byte space
      //     21 bytes Y direction cosine
      //      1 byte space
      //     21 bytes Z direction cosine
      //      2 bytes <CR><LF>
      //    -------
      //     92 bytes
      //
      //  for quaternions:
      //     28 bytes for quaternion values
      //      2 bytes <CR><LF>
      //  --------
      //     55 bytes
      //
      //  for orientation values:
      //     21 bytes orientation values
      //      2 bytes <CR><LF>
      //    -------
      //     48 bytes
      // ------------------------------------------------------------------
#define Cosines_Data_Size     92
#define Quaternions_Data_Size 55
#define Orientation_Data_Size 48

#define Error_Record_Size     14
```

```
#define Stations_Record_Size  13


#define Envelope_Data_Size 3
        // 3 bytes of data per max/min

      enum  Polhemus_read_type   { Cosines, Quaternions, Orientation };

      enum  Hemisphere_type      { X_axis, Y_axis, Z_axis           };

      enum  Station_type         { Station1, Station2, Station3, Station4,
                                   Station5, Station6, Station7, Station8 };

      struct Position_type       { float    x, y, z;  };
         // Data structure for the position info

      struct Direction_Cosine_type  { float x, y, z;  };
         // Data structure for the dir cosines

      struct Orientation_type       { float azimuth, elevation, roll;  };
         // Orientation data structure

      struct Quaternion_type        { float q0, q1, q2, q3;            };
         // Quaternion data structure

      struct Envelope_type          {
            byte Xmin[Envelope_Data_Size];  // Min X val of envelope
            byte Xmax[Envelope_Data_Size];  // Max "  "  "     "
            byte  Ymin[Envelope_Data_Size];  // Min Y  "  "     "
            byte Ymax[Envelope_Data_Size];  // Max "  "  "     "
            byte Zmin[Envelope_Data_Size];  // Min Z  "  "     "
            byte Zmax[Envelope_Data_Size];  // Max "  "  "     "
                                  };

   private:

#define Max_Stations           8
        // up to 8 stations may be used (depending on the hardware)

      boolean                 Station_Active[Max_Stations];
                              // Keeps a vector of the active
                              // stations.

      int                     num2read;   // Keeps track of the number of
                                          // bytes needed for a full data
                                          // packet.
      char                    Polhemus_data[100];  // The read in data

      boolean                 Polhemus_On;   // Keeps track if the Polhemus
                                             // device is turned on.
```

```
    Polhemus_read_type      Read_mode;  // Keeps track of the read mode
                                        // (what type of data is returned
                                        // from a read).
    Envelope_type           Envelope;   // Defines the operational
                                        // envelope for a station.

    Hemisphere_type         Hemisphere; // Defines the operational hemisphere
                                        // relative to the source.

    float                   Source_Height; // Height of the source off of
                                           // the floor.
    float                   scale_factor;

    int                     Data_length;   // Defines the total number of
                                           // bytes returned following a
                                           // poll request.

    boolean                 Read_in_progress;  // Indicates if there is an
                                               // unsatisfied poll request
                                               // pending.
    boolean                 port_is_open;      // Is the RS232 port open?
    int                     Reads_satisfied;   // Keeps track of the number
                                               // stations that have been
                                               // read during the read cycle
    int                     Active_Stations;   // Keeps track of the total
                                               // number of active stations

    void    count_active_stations();        // This method counts the
                                            // number of active stations.
    void    set_data_length();              // This method sets the num2read
                                            // attribute based on the number
                                            // of active stations and the
                                            // type of data being read
    boolean write_Polhemus(byte*, int);     // Private method for writing
                                            // to the Polhemus.

    Unmanaged_RS232_Port    *Pol_Port;      // The RS232 port handler


public:


    Polhemus_Class (port_numbers    port_num   = Port::port_two,
                    port_speed_spec port_speed = Port::b19200,
                    port_input_mode port_mode  = Port::raw,
                    Polhemus_read_type   mode  = Cosines)


    { // RS232_Port constructor called before following statements
```

```
   Pol_Port = new Unmanaged_RS232_Port (port_num,
                                        port_speed,
                                        port_mode,
                                        Port::terminal);
   for (int i = 0; i < Max_Stations; i++)
      Station_Active[i] = FALSE;

   Read_in_progress = FALSE;
   Reads_satisfied  =     0;

   Read_mode      = mode;
   // The Data_length attribute is initialized by initialize_Polhemus

   strcpy(Envelope.Xmax, " 65");
   strcpy(Envelope.Xmin, "-65");
   strcpy(Envelope.Ymax, " 65");
   strcpy(Envelope.Ymin, "-65");
   strcpy(Envelope.Zmax, " 65");
   strcpy(Envelope.Zmin, "  0");

   scale_factor  =  1.0;

   port_is_open  = Pol_Port->Open_Port();       // Open the port

   if (port_is_open == TRUE)
      Polhemus_On   = initialize_Polhemus(); // initializes all class
                                             // attributes.
   else
      fprintf (stderr, "Unable to open the Polhemus port!\n");
}


boolean read_Polhemus ( Station_type*,
                        matrixType,
                        char*);

boolean read_Polhemus ( Station_type*,
                        Position_type*,
                        Quaternion_type*,
                        char*);

boolean read_Polhemus ( Station_type*,
                        Position_type*,
                        Orientation_type*,
                        char*);

boolean read_Adjusted_Polhemus
                     ( Station_type*,
                       matrixType,
                       char*);
```

```
boolean read_Adjusted_Polhemus
                    ( Station_type*,
                      Position_type*,
                      Quaternion_type*,
                      char*);

boolean read_Adjusted_Polhemus
                    ( Station_type*,
                      Position_type*,
                      Orientation_type*,
                      char*);

boolean initialize_Polhemus   ();

boolean define_Allignment     (Station_type stat = Station1);

void    reset_Allignment ();

void    set_Polhemus_height   (float);

void    set_scale_factor      (float sf)
{ if (sf != 0.000)
  scale_factor = sf;
}

boolean set_Polhemus_boresight (Station_type = Station1,
                                boolean Wait = FALSE);

boolean set_Polhemus_read_mode (Polhemus_read_type,
                                boolean Wait = FALSE);

boolean set_Polhemus_envelope(Envelope_type,
                              Station_type stat = Station1,
                              boolean Wait = FALSE);

boolean set_Polhemus_hemisphere (Hemisphere_type,
                                 Station_type stat = Station1,
                                 boolean Wait = FALSE);

boolean set_Polhemus_to_inches (boolean Wait = FALSE);

boolean set_Polhemus_to_centimeters (boolean Wait = FALSE);

boolean reset_Polhemus_boresight (Station_type = Station1,
                                  boolean Wait = FALSE);

boolean reset_Polhemus ();

boolean activate_Polhemus_station   (Station_type  stat = Station2);

boolean deactivate_Polhemus_station (Station_type  stat = Station2);
```

```
        boolean poll_Polhemus ();

        void    Suspend_Input ()
        { Polhemus_On = FALSE; }

        void    Resume_Input ()
        { if (port_is_open == TRUE) Polhemus_On = TRUE; }

        ~Polhemus_Class ()
        {  reset_Polhemus ();
           Pol_Port->Flush_Queue();
           Pol_Port->Close_Port();
           Polhemus_On = FALSE;
           delete (Pol_Port);
        }

    };

#endif
```

```
// ------------------------------------------------------------------
// DATE:     12 Aug 91
// FILENAME: Socket_class.h
// VERSION:  1.0
// DESCRIPTION:
//    This file defines the Socket_class.  An instance of the Socket_class
//  can be used to establish a communications link with a remote machine
//  using sockets.
//    The client-server paradigm of operation is used:
//
//          Server:                     Client:
//
//          make_a_socket               make_a_socket
//               |                           |
//               V                           V
//   .-----> listen_for_connections      open_commlink
//   |           |                           /
//   |           |<-------------------------'
//   |           V
//   |       accept commlink ----------> write_commlink
//   |           |                         /  |
//   |           |<-----data--------------'   |
//   |           V                            |
//   |       read_commlink                    |
//   |           |                            |
//   |           V                            |
//   |       process the data                 |
//   |           |                            |
//   |           V                            |
//   |       write_commlink                   |
//   |           |   \                        |
//   |           |    '----data------> read_commlink
//   |           .                            |
//   |           .                            .
//   |           .                            .
//   '------ close_commlink                   .
//                                      close_commlink
//                                            .
//                                            .
//                                            .
// EXPORTED METHODS:
//    make_socket  -- This method creates a file descriptor for a socket.
//    open_commlink - This method is used to open a communications link
//                    to a remote host who is listening for connections
//                    at the specified port number.
//                    (Client side operation)
//    listen_for_connections -- This method is used to listen for
//                    connections at a specified port number.  Note that
//                    this is a blocking operation.  (Server side
//                    operation)
//    read_commlink -- This method is used to read data from a connected
```

```
//                    socket.
//    write_commlink -- This method is used to write data to a connected
//                    socket.
//    close_commlink -- This method discards any data pending on the port
//                    and closes the port.  Any attempt to read from or
//                    write to a closed commlink will result in an error.
//    get_max_queue_length -- This method returns the maximum depth of
//                    the port's connection queue.  (See get_max_queue_length
//                    for more information.  Server side operation.)
//    set_max_queue_length -- This method sets the maximum depth of the
//                    queue for processes waiting to connect to a specified
//                    port.  For example, if the maximum queue length is
//                    three, then up to three processes can be executing
//                    an open_commlink (or comparable) command for the
//                    same port on a remote system.  Any additional processes
//                    attempting to connect to the port will have their
//                    request refused.
//    get_port_info -- This method returns the service name of the
//                    specified port.  If no system service exists at
//                    that port, a null string is returned.
// AUTHOR:  Capt Mark Gerken
// ------------------------------------------------------------------------
#ifndef _SOCKETS_H_
#define _SOCKETS_H_




// ------------------------------------------------------------------------
// -------------           Exported Typedefs        ---------------------
// ------------------------------------------------------------------------


#ifndef host_type
#define host_type char*
#endif

#ifndef buffer_type
#define buffer_type char*
#endif

#ifndef socket_type
#define socket_type int
#endif

#ifndef network_port_type
#define network_port_type int
#endif

#ifndef boolean
```

```
#define boolean int
#endif

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif


// ---------------------------------------------------------------------
// --------------          Class Definition          ---------------------
// ---------------------------------------------------------------------




class Socket_class  {

   private:

         int   MaxQueueLength;  // the maximum length of the queue for processes
                                // waiting to connect to a given port number

   public:

      socket_type  make_socket            ( void );  // Gets a socket number

      socket_type  open_commlink          ( socket_type,
                                            host_type,
                                            network_port_type    );

      socket_type  listen_for_connections ( socket_type,
                                            network_port_type    );

      int          read_commlink          ( socket_type,
                                            buffer_type,
                                            int          );

      int          write_commlink         ( socket_type,
                                            buffer_type  );

      void         close_commlink         ( socket_type  );

      int          get_max_queue_length   ( void         )
      { return (MaxQueueLength); }

      void         set_max_queue_length   ( int  len     )
```

```
        { MaxQueueLength = len;    }

        char*        get_port_info          ( network_port_type    );

        Socket_class ()
        {
           MaxQueueLength = 3;
           set_max_queue_length(MaxQueueLength);
        }

           // use default destructor

};


#endif
```

```
// ------------------------------------------------------------------
// DATE:     16 Sep 91
// VERSION:  1.1
// NAME:     DGclient.h
// DESCRIPTION:
//     This class encapsulates the interface to the VPL DataGlove.
//  Note that the remote programs DGserver, DGinterface, and DG2P2server
//  must be loaded and running on louvre.   (Since the DGserver program
//  spawns the DGinterface program, which in turn spawns the DG2P2server
//  program, and since a process started by an rsh call to a remote
//  host is sterile (i.e., it cannot spawn children of its own), the
//  user of this class is responsible for starting the DGserver program).
//     Each user of this class should create their own DataGlove
//  calibration file.  This file permits the software to adjust
//  its interpretation of the DataGlove data to each individual's
//  hand.  A separate program, DG2cal (by SimmGraphics Engineering
//  Corporation) may be used to create the calibration file.  However,
//  this class also supports the creation of the calibration file.
//     Note that a separate program, DG2gest (by SimmGraphics
//  Engineering Corporation) is used for defining and building
//  the set of DatGlove gestures.  Once the DatGlove gesture file
//  has been created, this class can instruct the DG2P2server program
//  to read and load the gesture file.  Thereafter, the wearer of
//  the glove may form gestures, and the DG2P2server program can recognize
//  the gesture and identify it to the calling program by name and number.
//
//     This class is built on the client-server paridigm.  This class
//  encapsulate the client side of the relationship:
//
//      SG 4D Machine                      DEC (louvre)
//      ------------                       ----------
//     |            |------command------> |          |
//     | DGclient   |<----- data  ------- | DGserver |
//     |            |<-----status ------- |          |
//     '------------'                     '----------'
//                                             ^
//                                             |
//                                             v
//                                       -------------
//                                      |             |
//                                      | DGinterface |
//                                      '-------------'
//                                             ^
//                                             |
//                                             v
//                  -------------       -------------
//                 |     VPL     |     |             |
//                 | DataGLove   |<--->| DG2P2server |
//                 '-------------'     '-------------'
//
//   Note that three separate communication lines are used between the
```

```
// DGserver and the DGclient.  These three communications lines are
// implemented through sockets.  The communication line between the
// DGserver and the DGinterface is implemented using shared memory;
// shared memory is also used to implement the link between the
// DGinterface and the DG2P2server programs.  The link between the
// DG2P2server program and the VPL DataGlove is implemented using
// an RS232 cable.
//
// SUGGESTED ALGORITHM (for using the DG_class methods):
//     begin
//        sart the DGserver program
//        if no DataGlove calibration file exists for the user
//        then begin
//           set the path to the calibration file to be created
//              (set_cal_file_path)
//           create a calibration file (calibrate_dataglove)
//        end
//        else
//           set the path to the existing calibration file (set_cal_file_path)
//        end if
//        set the path to the gesture file (set_gest_file_path)
//        open the dataglove (open_dataglove)
//        loop
//            poll the dataglove (poll_dataglove)
//            render
//            read the DataGlove (read_dataglove)
//        end loop
//        close the DataGlove (close_dataglove)
//     end
//
// FILES READ:
//      <gesture_file> -- The file containing the gestures defined using
//                        the DG2gest program.  Note that the path to and
//                        the filename of this file is user specified;
//                        hovever, in the absence of a user specified file,
//                        the default file
//                         /usr/eng/mgerken/thesis/louvre/VE_default.cal
//                        will be used.
//      <calibration_file> -- This file contains user specific calibration
//                        information.  Either this class or the DG2cal
//                        program may be used to create this file.  Note
//                        that like the gesture_file, the user is responsible
//                        for identifying the name and location of the
//                        calibration file.  If none is specified, the default
//                        calibration file
//                          /usr/eng/mgerken/thesis/louvre/VE_default.gest
//                        will be used.
// FILES WRITTEN:
//     <calibration_file> -- Created by the calibrate_dataglove method. The
//                        user must specify the location and name of the
//                        created file.
```

```
// SUPPORTING CLASSES:
//     Socket_class -- used to open the communication links with the
//                     remote servers.
// EXPORTED METHODS:
//
//     set_cal_file_path  -- sets the path to and the name of the calibration
//                           file.
//     get_cal_file_path  -- returns the path to and the name of the
//                           calibration file
//     set_gest_file_path -- set the path to and the name of the gesture file.
//     get_gest_file_path -- returns the path to and the name of the gesture
//                           file
//     open_dataglove     -- This method opens the DataGlove and begins the
//                           flow of information from the device.
//     calibrate          -- This method creates a calibration file unique
//                           to the user.  Note that the calibration
//                           file name must be specified before the
//                           invocation of this method.
//     poll_dataglove     -- This method requests a datapacket from the
//                           DataGlove control unit.
//     read_dataglove     -- This method returns the name and number of the
//                           gesture formed, as well as a full set of DataGlove
//                           data.  If no valid gesture is formed,
//                           the name returned is NULL and the gesture
//                           number is 0.
//     read_dataglove_flex-- This method returns the name and number of the
//                           gesture formed, as well as the flexure values.
//                           If no valid gesture is formed,
//                           the name returned is NULL and the gesture
//                           number is 0.
//     orient_dataglove   -- This method affects the position and orientation
//                           of the DataGlove.
//     close_dataglove    -- This method shuts down the DataGlove.
//     DG_class           -- constructor.  This method sets the default values
//                           for the gesture file and configuration file paths.
//
// AUTHOR:   Capt Mark Gerken
// ------------------------------------------------------------

#ifndef _DG_CLIENT_
#define _DG_CLIENT_

#include "placemnt.h"
#include "Socket_class.h"
#include "DG_server_commands.h"
#include "DG2errs.h"

#ifndef Boolean
#define Boolean int
#endif
```

```
#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

    // Note that the below value is coupled to the SimmGraphics DG2lib.c
    // software.
#ifndef NUMSENSORS
#define NUMSENSORS 10
#endif

#define GESTURE_NAME_LENGTH 16
   //defined as 16 due to SimmGraphics software
#define POLHEMUS_UNITS_TO_INCHES  0.001998352  // 65/32767
#define POLHEMUS_ANGLE_TO_DEGREES 0.006994327

#define DEFAULT_GEST_FILE      "bm.gest"
#define DEFAULT_GEST_FILE_PATH "/usr/eng/mgerken/thesis/louvre"
#define DEFAULT_CAL_FILE       "bm.cal"
#define DEFAULT_CAL_FILE_PATH  "/usr/eng/mgerken/thesis/louvre"



      struct GestureData
      {   char          gesture;
          char          gesture_name[GESTURE_NAME_LENGTH];
      };

      struct SmallPosition {   float x, y, z; };

      struct Orientations  {   float yaw, pitch, roll;  };

      struct FlexData
      {   short         flex[NUMSENSORS];
      };

      struct DatagloveData
      {   SmallPosition pos;
          Orientations  orient;
          short         flex[NUMSENSORS];
      };


// ------------------------------------------------------------
// ------ ----------     CLASS DEFINITION FOLLOWS     ----------
// ------------------------------------------------------------

class DG_class : public Placement {
```

```
public:

   enum DG_read_type {full_record, flex_record};

private:

      // Supporting classes:
   Socket_class commlink;

      // Private attributes:
   DG_read_type  dg_read_mode;
   Boolean       poll_outstanding;
   Boolean       dg_initialized;
   Boolean       dg_opened;
   Boolean       dg_data_flowing;

   int  cmd_port;    // Network port numbers for the
   int  data_port;   // command, data, and status from
   int  status_port; // the DGserver.

   int  cmd_sock;    // The connected sockets for command,
   int  data_sock;   // data, and status.
   int  status_sock;

   int dg_data_length;
   int dg_bytes_left;

   char calibration_file[MAX_PATH_LENGTH];
   char gesture_file    [MAX_PATH_LENGTH];
   char DGbuf[126];   // stores the DG data as it's read in

   DatagloveData  dg_data;
   GestureData    gest_data;

      // Private methods:
   void parse_dataglove (DatagloveData  *dd,
                         GestureData    *gest,
                         char           *buf);

   int send_command ( int cmd, char *data);

   void get_dg_data_length ();


public:

   Boolean set_dg_read_mode  (DG_read_type r_type);

   int     set_cal_file_path (char  *cal_file);
   void    get_cal_file_path (char  *cal_file);
```

```
        int     set_gest_file_path (char *gest_file);
        void    get_gest_file_path (char *gest_file);

        int     open_dataglove (char *tty = "/dev/tty03");
        int     start_dataglove ();
        int     stop_dataglove  ();

        int     init_dataglove ();

        int     poll_dataglove ();

        int     read_dataglove (GestureData   *gest,
                                DatagloveData *dg = 0);
        int     read_dataglove_flex (GestureData   *gest,
                                     FlexData      *fd = 0);

        int     calibrate (char *new_cal_file);

        Boolean is_initialized ()
        { return (dg_initialized);   }

        Boolean is_open ()
        { return (dg_opened);        }

        Boolean data_is_flowing ()
        { return (dg_data_flowing);  }

        Boolean poll_pending()
        { return (poll_outstanding); }

        void      close_dataglove ();

        DG_class ( int cmd_port    = DEFAULT_CMD_PORT,
                   int data_port   = DEFAULT_DATA_PORT,
                   int status_port = DEFAULT_STATUS_PORT);

        ~DG_class()
        {
           close_dataglove();
           send_command (TERMINATE, "");
           commlink.close_commlink (data_sock);
           commlink.close_commlink (status_sock);
           commlink.close_commlink (cmd_sock);
        }

};
#endif
```

```
// ------------------------------------------------------------
// DATE:    20 Oct 91
// VERSION: 1.0
// NAME:    Event.h
// DESCRIPTION:
//
// FILES WRITTEN:     None
// EXPORTED METHODS:
//     get_event : this method analyzes the VPL DataGlove and
//                 the Polhemus tracker data for the occurrence
//                 of specific events.  The number of the event
//                 recognized is returned to the calling procedure.
//                 (Note that the NULL event is considered a valid
//                 event.  The NULL event is returned if no other
//                 valid event is recognized.)
//     begin_watch_for_event (event) : This method is used to initialize
//                 the parameters required to recognize events
//                 involving rotations or translations.  For example,
//                 if the WRIST_TWIST is an event of interest, then
//                 this method should be called with the argument
//                 WRIST_TWIST at the point in the program where the
//                 monitoring of the event should begin.   The events
//                 involving rotations or translations are:
//                       SAFE
//                       CUT
//                       WRIST_TWIST
//                 Note that the WRIST_TWIST  event is triggered once
//                 the wrist has been rotated more than 60 degrees from
//                 the orientation of the wrist at the time when the
//                 begin_watch_for_event method was called.
//     end_watch_for_event (event) : Terminates the watch for an event
//                 involving rotations or translations.  This method has
//                 no effect on non-rotational or non-translational events.
//
// AUTHOR:  Capt Mark Gerken
// ------------------------------------------------------------

#include "globals.h"

class Event_Recognizer_Class {

   private:

      struct position {
                        float x;
                        float y;
                        float z;
                      };

      float    initial_twist_angle;        // for wrist_twist
      float    initial_cut_angle;          // for cut
```

```
    position old_position;              // for safe

    boolean  cut_active;
    boolean  cut_initialized;
    boolean  wrist_twist_active;
    boolean  wrist_twist_initialized;
    boolean  use_DYj_for_twist;
    boolean  safe_active;
    boolean  safe_initialized;

    boolean  cut_recognized  (float   DXk,
                              float   DYj,
                              char   *posture_name);

    boolean  wrist_twist_recognized (float   DXk,
                                     float   DYj,
                                     float   DYk);

    boolean  safe_recognized (float xpos,
                              float ypos,
                              float zpos,
                              char   *posture_name);
public:

    int get_event (matrixType   PolData,
                   char   *posture_name);

    void  begin_watch_for_event (int          event);

    void  end_watch_for_event   (int          event);

    Event_Recognizer_Class ()
    {  cut_active              = FALSE;
       cut_initialized         = FALSE;
       safe_active             = FALSE;
       safe_initialized        = FALSE;
       use_DYj_for_twist       = FALSE;
       wrist_twist_active      = FALSE;
       wrist_twist_initialized = FALSE;
       old_position.x = 0.0;
       old_position.y = 0.0;
       old_position.z = 0.0;
    }

    // default destructor
};
```

```
// ------------------------------------------------------------
// DATE:    20 Oct 91
// VERSION: 1.0
// NAME:    FSMclass.h
// DESCRIPTION:
//    This header file defines the finite state machine (FSM) class
// used by the Virtual World system.
//
// FILES READ:     FSMfile.  This file defines the valid state
//   transitions, and the actions associated with those transitions.
//   The general format of the file is:
//       from_state  event  to_state  action
//   where from_state is an integer identifying the state to which
//   the transition originates.  Event identifies under what circumstances
//   the transition is to be made (for instance, when the user forms
//   the "horns" posture).  The to_state field defines to destination
//   state for the transition, and action identifies the system call
//   to make during the transition.
//
// FILES WRITTEN:     None
// SUPPORTING CLASSES:
//    Event_Recognizer_Class contained within the FSM for performing
//              the task of event recognition.
// EXPORTED METHODS:
//    FSMclass : Constructor.  This method takes an optional parameter
//              identifying the FSMfile.  The constructor processes the
//              FSMfile and creates the corresponding FSM.
//    process_event:  This method accepts and processes events, and is
//              responsible for updating the current state of the FSM
//              and for identifying the resulting system action.  Note
//              that some system events (such as wrist_twist) require
//              some sort of trigger;  process_events will provide this
//              trigger.
//    ~FSMclass : Destructor.
//
// AUTHOR:  Capt Mark Gerken
// ------------------------------------------------------------



#include "Event.h"
#include "EventDefs.h"


#define  MAX_STATES      100   // No more than 100 total states
#define  MAX_TRANSITIONS  10   // No more than 10 transitions/state
#define  NOT_USED         -1


class FSM_Class {
```

```
private:

   Event_Recognizer_Class ERC;  // This class recognizes events
                                // used for making the FSM state
                                // transitions.

   int Current_state;

      // Note that the actions are associated with the
      // transitions, not the states!
   struct Transition_struct {
                              int Event;
                              int To_State;
                              int Action;
                            };

      // Each state has a maximum of 10 valid transitions from it
   struct States {
                  int                Num_def_trans;   // The number of
                                                      // defined transitions
                  Transition_struct  Defined_transitions[MAX_TRANSITIONS];
                  };

   States  State_Table[MAX_STATES];

   int    define_transition (int   from_state,
                             int   event,
                             int   to_state,
                             int   action);

   int    convert_event_string  (char *event);

   int    convert_action_string (char *action);

public:

   void   reset ()
   {
      ERC.end_watch_for_event (CUT);
      ERC.end_watch_for_event (SAFE);
      ERC.end_watch_for_event (WRIST_TWIST);
      Current_state = 0;
      State_Table[Current_state].Num_def_trans);
      for (int t = 0; t < State_Table[Current_state].Num_def_trans; t++)
      {
         State_Table[Current_state].Defined_transitions[t].Event,
         State_Table[Current_state].Defined_transitions[t].To_State,
         State_Table[Current_state].Defined_transitions[t].Action);

         if (State_Table[Current_state].Defined_transitions[t].Event == CUT)
            ERC.begin_watch_for_event (CUT);
```

```
            if (State_Table[Current_state].Defined_transitions[t].Event == SAFE)
               ERC.begin_watch_for_event (SAFE);

            if (State_Table[Current_state].Defined_transitions[t].Event == WRIST_TWIST)
               ERC.begin_watch_for_event (WRIST_TWIST);
         }
      }


      int    update_state (matrixType PolhemusMatrix,
                           char      *posture_name);

      FSM_Class (char  *FSMfile_name);

      ~FSM_Class ()
      { reset(); }

};
```

Appendix C. *Definition of the Finite State Machine for the BattleManager System*

This appendix contains the definition of the finite state machine used in support of the BattleManager program, and describes the file format for the tranition table information.

The file format for the transition table is quite simple. Each line in the file consists of four entries defining a single transition:

- The current state. This integer entry identifies the state from which the transition originates.
- An input symbol. This entry identifies the input symbol which will trigger the transition from the current state. (The set of input symbols is defined in Section 3.7.2.)
- An output symbol. This entry identifies the system action requested during the transition from the current state to the next state. (The output symbols are defined in Section 3.7.3.)
- The next state. This integer entry identifies the state into which the finite state machine will transition following the receipt of the specified input symbol. Note that the next state and the current state may be the same.

Comments may be included in the file provided they are preceeded with a double slash (*i. e.*, a "//"). The transition table I created in support of the BattleManager program is located the end of the appendix.

The transition diagrams, called *Markov state diagrams*, provide a pictoral representation of the finite state machine. Figure C.1 depicts the transitions into and out of the initial state. In the current implmentation, the initial state of the finite state machine (the q_0 state) is state 0. The remaining figures depict the Markov diagrams corresponding to seven of the eight transitions originating from the initial state.

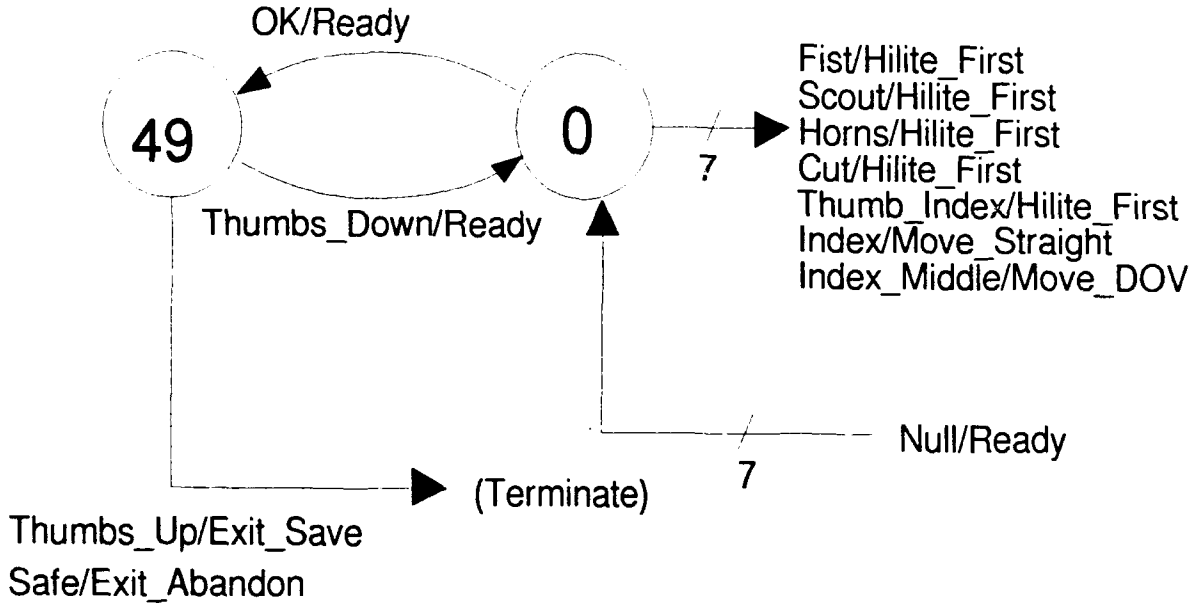


Figure C.1. Transitions out of the Initial State

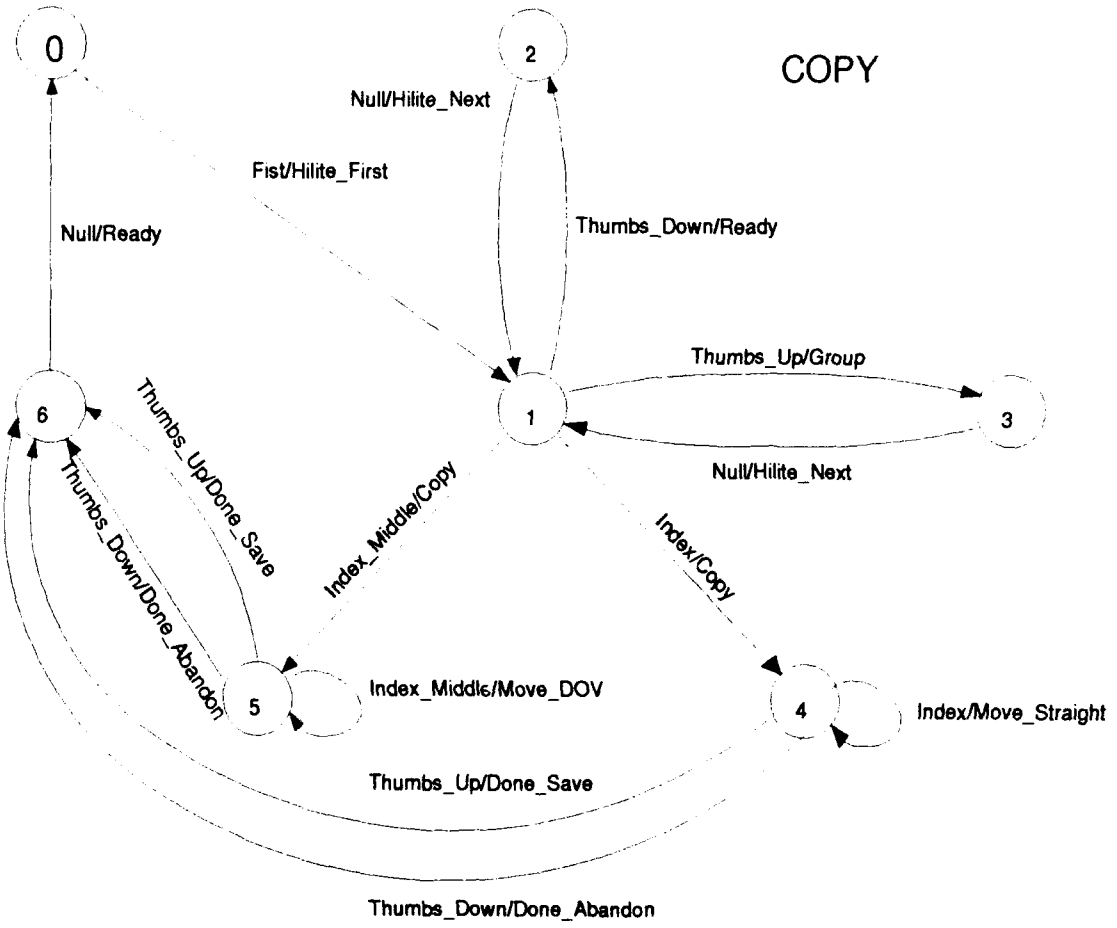


Figure C.2. Markov State Diagram for Copy

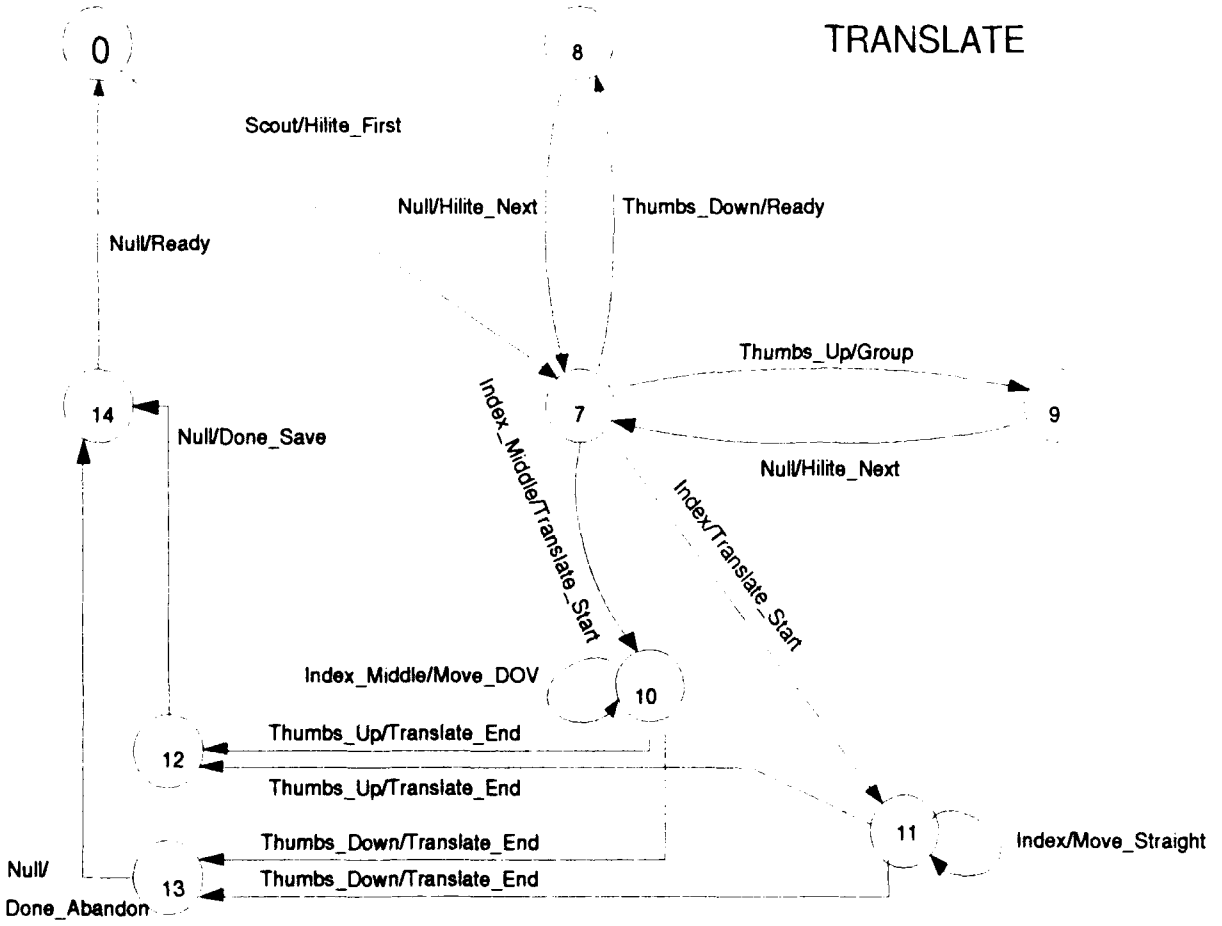


Figure C.3. Markov State Diagram for Translate

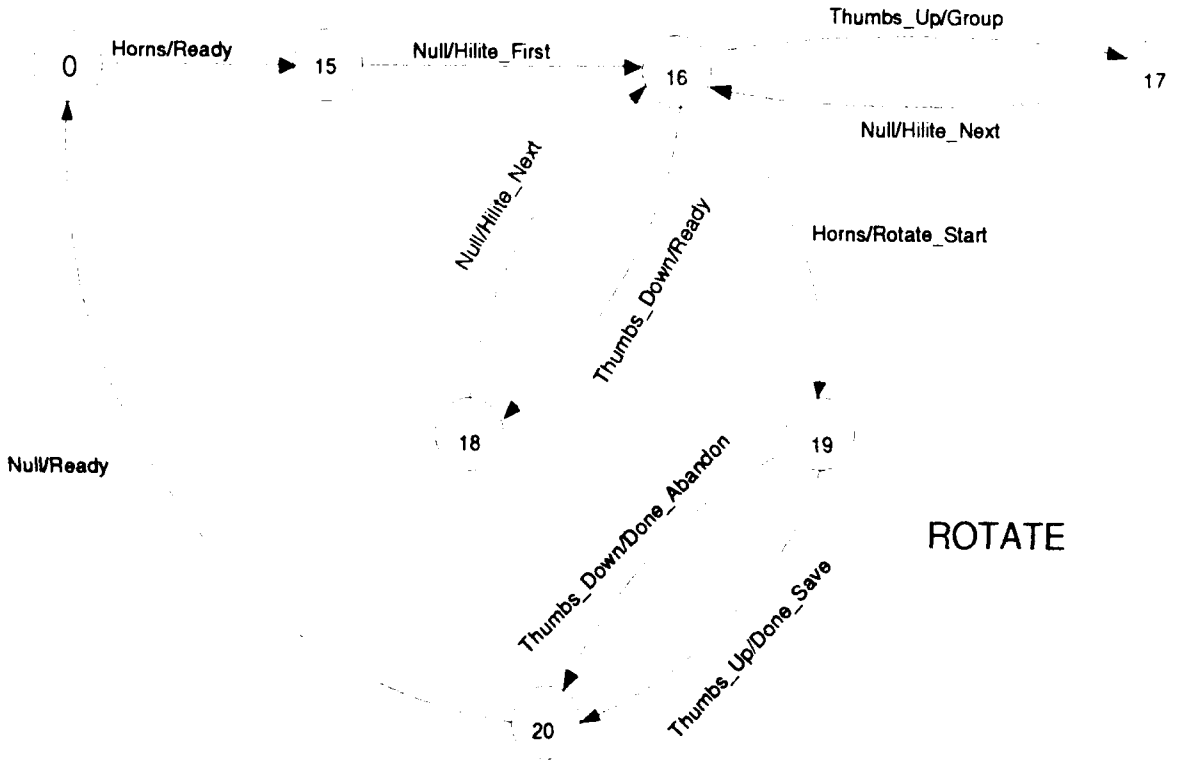


Figure C.4. Markov State Diagram for Rotate

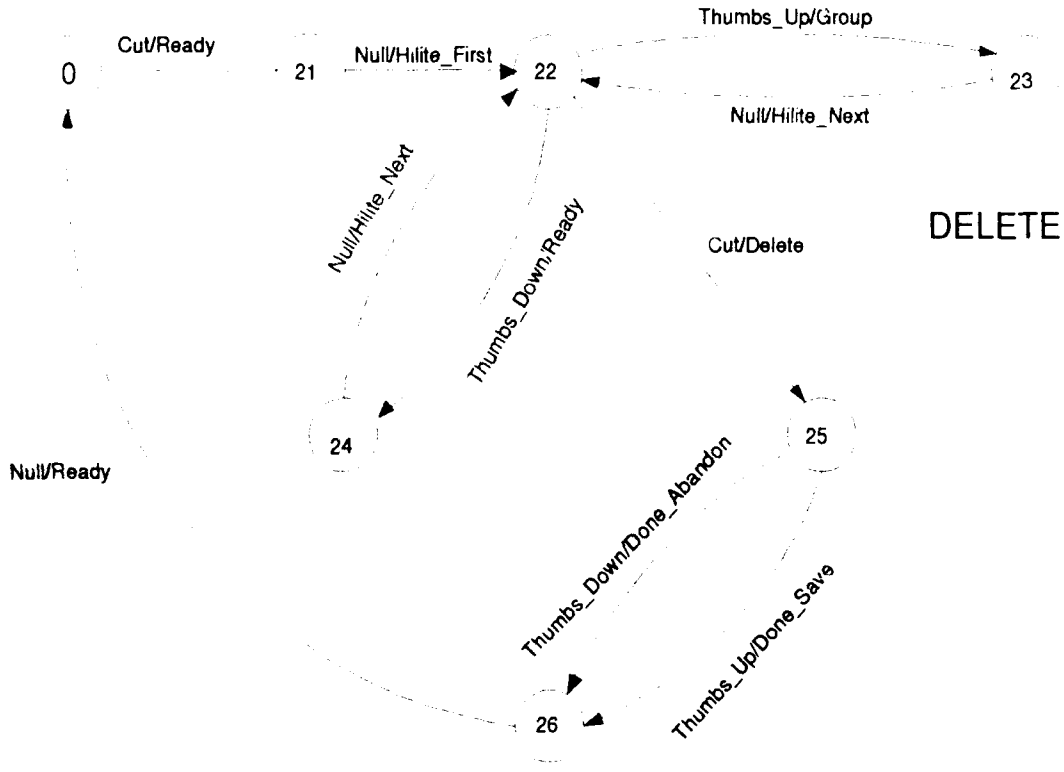


Figure C.5. Markov State Diagram for Delete

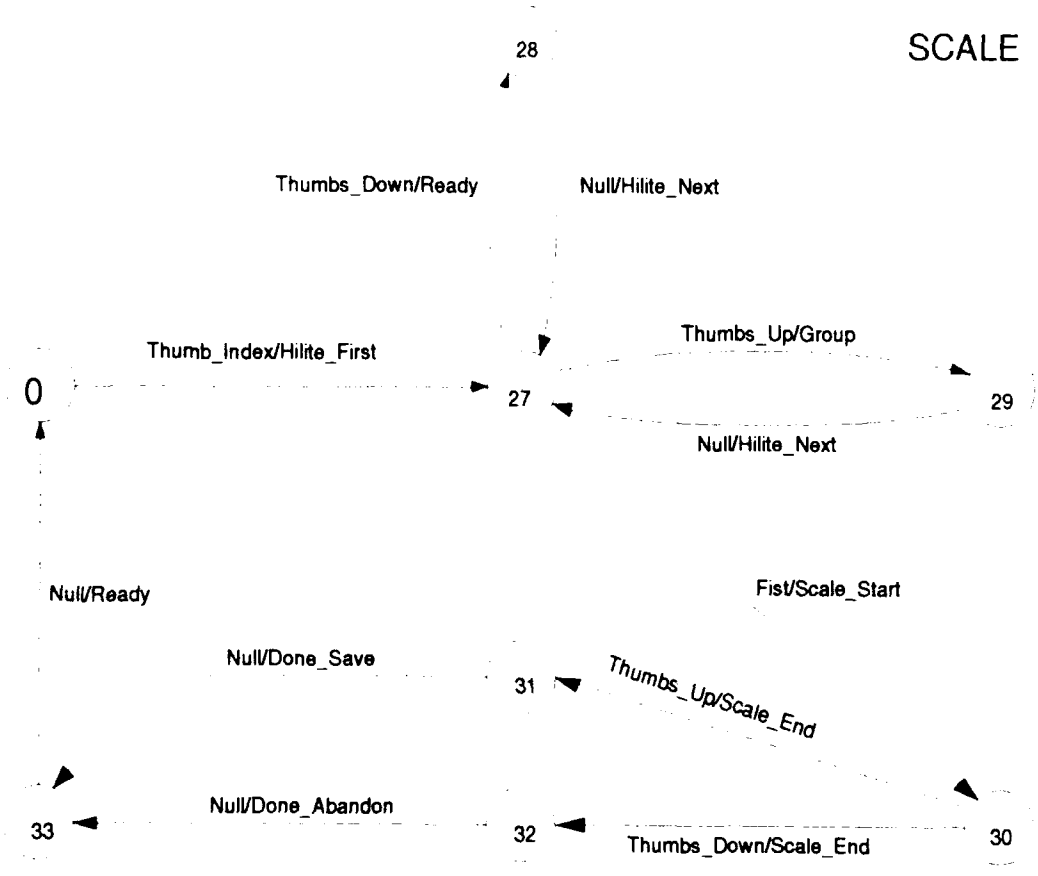


Figure C.6. Markov State Diagram for Scale

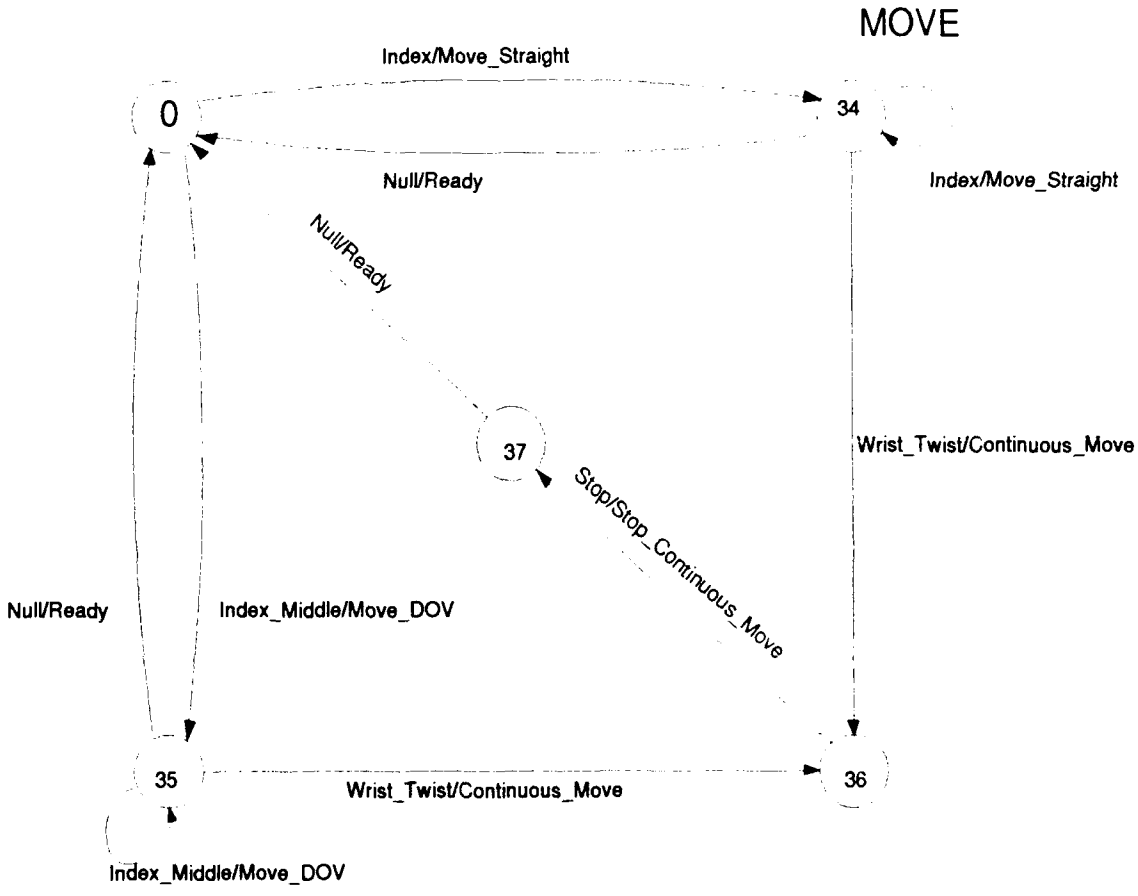


Figure C.7. Markov State Diagram for Moving the Eyepoint

```
0               OK              READY           49 RED
49              THUMBS_DOWN     READY            0 DEFAULT
49              THUMBS_UP       EXIT_SAVE       50 WHITE
49              SAFE            EXIT_ABANDON    50 BLACK
// COPY
0               FIST            HILITE_FIRST     1 BLUE
1               THUMBS_DOWN     READY            2 BLACK
2               NULL            HILITE_NEXT      1 BLUE
1               THUMBS_UP       GROUP            3 WHITE
3               NULL            HILITE_NEXT      1 BLUE
1               INDEX           COPY             4 GREEN
4               INDEX           TRANSLATE_START 11 GREEN
4               THUMBS_UP       DONE_SAVE        6 WHITE
4               THUMBS_DOWN     DONE_ABANDON     6 BLACK
1               INDEX_MIDDLE    COPY             5 GREEN
5               INDEX_MIDDLE    TRANSLATE_START 10 GREEN
5               THUMBS_UP       DONE_SAVE        6 WHITE
5               THUMBS_DOWN     DONE_ABANDON     6 BLACK
6               NULL            READY            0 DEFAULT
// TRANSLATE
0               SCOUT           HILITE_FIRST     7 GRAY
7               THUMBS_DOWN     READY            8 BLACK
8               NULL            HILITE_NEXT      7 GRAY
7               THUMBS_UP       GROUP            9 WHITE
9               NULL            HILITE_NEXT      7 GRAY
7               INDEX           TRANSLATE_START 11 GREEN
7               INDEX_MIDDLE    TRANSLATE_START 10 GREEN
10              INDEX_MIDDLE    MOVE_DOV        10 GREEN
10              THUMBS_UP       TRANSLATE_END   12 WHITE
10              THUMBS_DOWN     TRANSLATE_END   13 BLACK
11              INDEX           MOVE_STRAIGHT   11 GREEN
11              THUMBS_UP       TRANSLATE_END   12 WHITE
11              THUMBS_DOWN     TRANSLATE_END   13 BLACK
12              NULL            DONE_SAVE       14 WHITE
13              NULL            DONE_ABANDON    14 BLACK
14              NULL            READY            0  DEFAULT
// ROTATE
0               HORNS           HILITE_FIRST    15 CYAN
15              NULL            READY           16 CYAN
16              THUMBS_UP       GROUP           18 WHITE
16              THUMBS_DOWN     READY           18 BLACK
18              NULL            HILITE_NEXT     16 CYAN
16              HORNS           ROTATE_START    17 CYAN
17              NULL            ROTATE_END      19 YELLOW
19              HORNS           ROTATE_START    17 CYAN
```

```
19              THUMBS_DOWN     DONE_ABANDON            20 BLACK
19              THUMBS_UP       DONE_SAVE               20 WHITE
20              NULL            READY                   0  DEFAULT
// DELETE
0               CUT             READY                   21 YELLOW
21              NULL            HILITE_FIRST            22 YELLOW
22              THUMBS_UP       GROUP                   23 WHITE
23              NULL            HILITE_NEXT             22 YELLOW
22              THUMBS_DOWN     READY                   24 BLACK
24              NULL            HILITE_NEXT             22 YELLOW
22              CUT             DELETE                  25 RED
26              THUMBS_UP       DONE_SAVE               26 WHITE
25              THUMBS_DOWN     DONE_ABANDON            26 BLACK
26              NULL            READY                   0  DEFAULT
// SCALE
0               THUMB_INDEX     HILITE_FIRST            27 MAGENTA
27              THUMBS_DOWN     READY                   28 BLACK
28              NULL            HILITE_NEXT             27 MAGENTA
27              THUMBS_UP       GROUP                   29 WHITE
29              NULL            READY                   27 MAGENTA
27              FIST            SCALE_START             30 MAGENTA
30              THUMBS_UP       SCALE_END               31 WHITE
30              THUMBS_DOWN     SCALE_END               32 BLACK
31              NULL            DONE_SAVE               33 WHITE
32              NULL            DONE_ABANDON            33 BLACK
33              NULL            READY                   0  DEFAULT
// MOVE
0                NDEX           MOVE_STRAIGHT           34 GREEN
34              NULL            READY                   0  DEFAULT
34              INDEX           MOVE_STRAIGHT           34 GREEN
34              WRIST_TWIST     CONTINUOUS_MOVE         36 GREEN
0               INDEX_MIDDLE    MOVE_DOV                35 GREEN
35              NULL            READY                   0  DEFAULT
35              WRIST_TWIST     CONTINUOUS_MOVE         36 GREEN
35              INDEX_MIDDLE    MOVE_DOV                35 GREEN
36              THUMBS_UP       SPEED_UP                36 GREEN
36              THUMBS_DOWN     SLOW_DOWN               36 GREEN
36              STOP            STOP_CONTINUOUS_MOVE    37 DEFAULT
37              NULL            READY                   0  DEFAULT
```

Bibliography

1. A. C. Louie. *Perspective 3-D Display Development for Command and Control Application*. Technical Report Report NOSC/TR-944, San Diego: Naval Ocean Systems Center, April 1984 (AD-B082 062L).

2. Ascension Technology Corporation, "A Flock of Birds: Product Description for High Speed Tracking of Multiple Bird Receivers and Operation of Multiple Receiver/Transmitter Configurations." Product Announcement, April 1991. Burlington VT.

3. Ascension Technology Corporation, "The Bird." Product Announcement, undated. Burlington VT.

4. Bishop, G. and H. Fuchs. "The Self-Tracker: A Smart Optical Sensor on Silicon." In *Proc 1984 MIT Conf on Advanced Research in VLSI*, pages 65–73, Dedham Mass: Artech House, 1984.

5. Booch, Grady. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, 1991.

6. Brooks, Jr., Frederick P. "UNC Force Display Research." In Zeltzer, D., editor, *ACM SIGGRAPH '89 Course Notes: Implementing and Interacting with Real Microworlds*, August 1989. Course 29.

7. Brooks, Jr., Frederick P., et al. "Project GROPE — Haptic Displays for Scientific Visualization." In *Proceedings of the ACM SIGGRAPH*, pages 177–185, 6-10 Aug 1990.

8. Brunderman, Capt John. *Design and Application of an Object Oriented Graphical Database Management System for Synthetic Environments*. MS thesis, AFIT/GA/ENG/91D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1991.

9. Chung, J. C. "Exploring Virtual Worlds with Head Mounted Displays." In *Proceedings of the SPIE: Non-Holographic True Three-Dimensional Displays*, Volume 1083, pages 42–52, January 1989.

10. Clapp, Robert E. "Stereoscopic Perception." In *SPIE: True 3D Imaging Techniques and Display Technologies*, Volume 761, pages 79–84, 1987.

11. Clark, J. H. "Designing Surfaces in 3-D." *Communications of the ACM*, *19*(8):454–460 (1976).

12. Dougherty, Edward R. and Charles R. Gardina. *Mathematical Methods for Artificial Intelligence and Autonomous Systems*. Prentence Hall, 1988.

13. Duckett, Capt Donnald T. *The Application of Satistical Estimation Techniques to Terrain Modeling*. MS thesis, AFIT/GCE/ENG/91D-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1991.

14. EXOS, Inc., "The Dexterous Hand Master." Product Announcement, undated. 8 Blanchard Road, Burlington MA.

15. Filer, Capt Robert E. *A 3-D Virtual Environment Display System.* MS thesis, AFIT/GCS/ENG/89D-2, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1989.

16. Fisher, S. S., et al. "Virtual Environment Display System." In *Proceedings of the ACM 1986 Workshop on Interactive 3-D Graphics*, pages 77–87, 1986.

17. Foley, James D., et al. *Computer Graphics: Principles and Practice*. Addison-Wessley Publishing Company, 1990.

18. GEC Ferranti, Gaertner Research Division, "GRD-1010 Position Tracking System for Simulation Applications." Product Announcement, undated. 140 Water Street, Nowalk Connecticut.

19. Green, Mark. "A Survey of Three Dialogue Models," *ACM Transactions on Graphics*, 5(3):224–275 (July 1987).

20. Harris, Frank E. *Preliminary Work on the Command and Control Workstation of the Future*. MS thesis, Naval Postgraduate School, Monterey CA, June 1988 (AD-A201 025).

21. Holloway, R. L. *Head-Mounted Display Technical Report*. Technical Report Report TR87-015, University of North Carolina Department of Computer Science, June 1987.

22. Iwata, Hiroo. "Artificial Reality with Force-Feedback: Development of Desktop Virtual Space with Compact Master Manipulator." In *Proceedings of the ACM SIGGRAPH*, pages 165–170, 1990.

23. Kanko, Capt Mark A. *Geometric Modeling of Flight Information for Graphical Cockpit Display*. MS thesis, AFIT/GCE/ENG/87D-6, School of Engineering, Air Force Institute of Techology (AU), Wright Patterson AFB OH, December 1987.

24. Kaufman, Arie, et al. "Direct Interaction with a 3D Volumetric Environment." In *Proceedings of the ACM SIGGRAPH*, pages 33–34, 1990.

25. Lorimor, Gary K. *Real-Time Display of Time Dependent Data Using a Head-Mounted Display*. MS thesis, AFIT/GE/ENG/88D-22, School of Engineering, Air Force Institute of Techology (AU), Wright Patterson AFB OH, December 1988.

26. Minsky, Margaret, et al. "Feeling and Seeing: Issues in Force Display." In *Proceedings of the ACM SIGGRAPH*, pages 235–243, 1990.

27. Olson, Capt Robert. *Techniques to Enhance the Visual Realism of a Synthetic Environment Flight Simulator*. MS thesis, AFIT/GCS/ENG/91D-16, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1991.

28. Polhemus Navigation Sciences. *3Space Users Manual*. Colchester VT, 1985.

29. Rebo, Capt Robert K. *A Helmet-Mounted Virtual Environment Display System*. MS thesis, AFIT/GCS/ENG/88D-17, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1988.

30. Rome Air Development Center, Griffis AFB NY: RADC/COE. *SCENARIO User's/Operator's Manual*, June 1989. Prepared by IIT Research Institute, Beeches Technical Campus, Route 26 North, Rome NY.

31. Silicon Graphics Inc. *IRIX System Administrator's Reference Manual*, 1989. Special Files Section, termio Entry.

32. Simpson, Capt Dennis J. *An Application of the Object-Oriented Paradigm to a Flight Simulator*. MS thesis, AFIT/GCS/ENG/91D-22, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1991.

33. StereoGraphics Corporation, "Crystal Eyes Stereo Viewing System." Product Announcement, 1991. San Raphael CA.

34. Sturman, David J., et al. "Hands-On Interaction with Virtual Environments." *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 19–24 (13-15 November 1989).

35. Sutherland, I. E. "The Ultimate Display." In *Information Processing 1965, Proceedings of the IFIP Congress 65*, pages 506–508, 1965.

36. Sutherland, I. E. "A Head-Mounted Three-Dimensional Display." In *AFIPS Conference Proceedings*, Volume 33, pages 757–764, 1968.

37. Textronix Inc., "Display Products: From 3-D to Stereo." Product Announcement, 1991.

38. UNC at Chapel Hill, "A Mountain Bike with Force Feedback for Indoor Exercise." Tomorrow's Realities pamphlet. Published by ACM SIGGRAPH, 29 July - 2 August 1991.

39. Veron, H., et al. *3D Displays for Battle Management*. Technical Report Report MTR-10689, Contract F19628-89-C-0001. Bedford MA: MITRE Corporation, April 1990 (AD-A223 142).

40. Vickers, D. L. *Sorcerers Apprentice: Head Mounted Display and Wand*. PhD dissertation, Department of Computer Science, University of Utah, 1974.

41. VPL Research, Inc., "Dataglove Model 2." Product Announcement, Mar 1989. Redwood City CA.

42. VPL Research, Inc., "VPL Virtual Reality Products." Product Announcement, 1991. Redwood City CA.

43. VPL Research Inc. "New DataGloves Include Tactile and Force Feedback." *Virtual World News*, *3*(1):1–2 (Summer 1991).

44. Wardin, Capt Charles L. *Battle Management Visualization System*. MS thesis, AFIT/GE/ENG/89D-56, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB Dayton OH, December 1989.

45. Weimer, David and S. K. Ganapthy. "A Synthetic Visual Environment with Hand Gestering and Voice Input." In *CHI'89 Conference Proceedings*, pages 235–240, April 1989.

46. Wright, Jeff. "Altered States: a software developer's vision of the future of virtual reality," *Computer Graphics World*, *12*(12):77+ (December 1989).

47. Wrightman, Frederick and Doris J. Kistler. *Field Measurement of Head Related Transfer Functions*. Technical Report, Madison WI: Wisconsin University, 1990 (AD-A227850).

Vita

Captain Mark Gerken was born on 30 January 1963 in Des Moines, Iowa. He graduated from Dowling High School in West Des Moines, Iowa in 1981. In February 1985 he married Debora Sue Mallory. He received a Bachelor of Science degree in Computer Engineering from Iowa State University in May of the following year. After receiving his commission through ROTC, he was assigned to 82nd Student Squadron at Williams AFB, Arizona, where he attended flight training. He was later assigned to the Directorate of Engineering Reliability within the Acquisition Logistics Divsion at Wright-Patterson AFB Ohio. He then transfered to the C-17 System Program Office at Wright-Patterson AFB where he remained until entering AFIT as a full-time student in May 1990. Captain Gerken has three children: Mallory (age five), Mark (age 3), and Meghan (age 1).

Permanent address: 5214 Cobb Drive
Dayton, Ohio 45431